Das MATE Projekt – visuelle Spezifikation von MATLAB Simulink/Stateflow Analysen und Transformationen

Ingo Stürmer[1], Heiko Dörr[2], Holger Giese[3], Udo Kelter[4], Andy Schürr[5], Albert Zündorf[6],

[1]Model Engineering Solutions, stuermer@acm.org
[2]DaimlerChrysler AG, heiko.doerr@daimlerchrysler.com
[3]Universität Paderborn, hg@uni-paderborn.de*
[4]Universität Siegen, kelter@informatik.uni-siegen.de
[5]TU Darmstadt, andy.schuerr@es.tu-darmstadt.de
[6]Universität Kassel, zuendorf@uni-kassel.de

Abstract: Die modellbasierte Entwicklung beginnt sich als Standardparadigma in der Steuergerätesoftwareentwicklung zu etablieren. Um die Wirksamkeit und Effizienz der modellbasierten Entwicklung zu erhöhen, sind Richtlinien für die Modellierung unerlässlich. Diese manuell zu überprüfen ist aufwändig und fehleranfällig. Das Projekt MATE (MATLAB Simulink/Stateflow Analysis and Transformation Environment) hat sich deshalb zum Ziel gesetzt, die bislang meist manuelle Prüfung der Einhaltung bzw. Verletzung von Modellierungsrichtlinien zu automatisieren, sowie vor allem Modelltransformationen zur Korrektur entdeckter Mängel und zur Unterstützung von Entwicklungsschritten und die Visualisierung der Unterschiede verschiedener Entwicklungsstände zu untersuchen.

1 Einleitung

Kennzeichnend für die modellbasierte Entwicklung eingebetteter Software im Automobil ist die frühzeitige Beschreibung der Regelungs- und Steuerungsalgorithmen durch ausführbare Modelle unter Verwendung von Blockschaltbildern und Zustandsübergangsdiagrammen. Hierbei wird der Ingenieur unterstützt durch Modellierungs- und Simulationswerkzeuge wie *MATLAB Simulink/Stateflow* [MW06]. Die zunehmende Komplexität der elektronischen Systeme im Kraftfahrzeug schlägt sich entsprechend in den Modellen nieder. Modellierungswerkzeuge bieten jedoch nur bedingt Mechanismen zur Beherrschung dieser Komplexität und zur Unterstützung der Entwickler. Besonders deutlich zeigen sich diese Probleme an (1) der bisher nicht hinreichend automatisierten Unterstützung der Überprüfung dutzender von Modellierungsrichtlinien und der Korrektur des Modells entsprechend der Regeln, (2) der mangelhaften Versionierungsunterstützung und der Darstellung von Unterschieden zwischen Modellen, und (3) der immer noch unzureichenden Integration externer Werkzeuge, z.B. Werkzeuge des Requirements Engineering, mit der Modellierungsumgebung (etwa in Punkten wie der bidirektionalen Propagation von Änderungen und der Verwaltung von Konsistenzbeziehungen zwischen Entwicklungsartefakten, die in verschiedenen Werkzeugen gespeichert sind).

* Augenblicklich Lehrstuhlvertreter am Hasso-Plattner-Institut der Universität Potsdam

Im Rahmen des Projekts *MATE* (*M*ATLAB Simulink/Stateflow *A*nalysis and *T*ransformation *E*nvironment) untersuchen sechs Projektpartner (siehe Autorenliste), wie mit Hilfe von Modellanalyse- und Transformationstechniken die oben genannten Probleme bei der modellbasierten Entwicklung gelöst werden können. Dabei spielen die Modellierungs- und Modelltransformationsumgebung *Fujaba* [Fuja06] sowie die darauf aufbauende Metamodellierungsumgebung *MOFLON* [MOF06] eine herausragende Rolle für die Spezifikation und Generierung von Zugriffschnittstellen, Modell-Repositories, Analysefunktionen und Editieroperationen für Matlab Simulink/Stateflow-Modellen. Fujaba und MOFLON bieten hierfür eine Vereinigung der (Meta-)Modellierungsstandards der OMG mit Graphtransformationen als präzise definiertem visuellem Spezifikationsansatz an.

Ziel des Projektes MATE ist es, eine eng mit den Werkzeugen MATLAB Simulink und Stateflow sowie dem darauf aufbauenden *Simulink ModelAdvisor* [MW06] integrierte Unterstützung folgender Entwicklungsaktivitäten (Anwendungsszenarien) anzubieten:

1. die Überprüfung von Modellierungsrichtlinien (durch Suche nach Anti-Patterns)
2. die Berechnung von Metriken (für Identifikation zu überarbeitender Modellteile)
3. die Generierung von Reparaturvorschlägen für Richtlinienverletzungen
4. die batchorientierte Durchführung vorgeschlagener Reparaturen von Richtlinienverletzungen durch Modelltransformationen
5. die Unterstützung komplexer interaktiver Editieroperationen (Einsatz empfohlener Entwurfsmuster, interaktive Elimination von Anti-Entwurfsmustern etc.)
6. sowie die Berechnung und Visualisierung der Unterschiede von Modellversionen

Die in der Praxis ebenfalls erwünschte Integration von MATLAB Simulink/Stateflow-Modellen mit anderen Entwicklungsartefakten (wie etwa mit in DOORS [Tel06] verwalteten Anforderungen) wurde im MATE-Projekt zunächst ausgeklammert, da die Integration der Daten verschiedener Werkzeuge sowie die Verwaltung bidirektionaler Traceability-Links Schwerpunkt des separaten Projekts *ToolNet* [ADS02] ist, das von einer Teilmenge der MATE-Partner durchgeführt wird.

Das Papier ist im Weiteren wie folgt gegliedert: In Abschnitt 2 wird zunächst die Spezifikation und Erkennung von Mustern erläutert, die die Basis für die Berechnung von Modellmetriken, die Überprüfung von Modellierungsrichtlinien und die Elimination von Richtlinienverletzungen in MATE bildet. Darauf aufbauend wird in Abschnitt 3 beschrieben, wie alle Arten von Änderungsoperationen auf MATLAB Simulink/Stateflow-Modellen in Form von Graphtransformationen spezifiziert werden. Die Realisierung der MATE-Werkzeugumgebung wird dann in Abschnitt 4 knapp skizziert. Den Abschluss des Beitrags bilden ein Vergleich mit verwandten Arbeiten in Abschnitt 5 und die üblichen Schlussbemerkungen in Abschnitt 6.

Damit werden von den oben aufgeführten und von MATE unterstützten Anwendungsszenarien die Spezifikation von Metriken sowie die Berechnung und Visualisierung von Modelldifferenzen in diesem Beitrag nicht ausführlich vorgestellt. Insbesondere bezüglich der Berechnung der Unterschiede von Modelldifferenzen sei der Leser deshalb auf weiterführende Darstellungen und Literatur wie [KWN05, SiDiff] hingewiesen.

2 Mustererkennung

Es hat sich gezeigt, dass die Einhaltung von Modellierungsrichtlinien und die Verwendung standardisierter Muster (Pattern) für die erfolgreiche Umsetzung von Anforderungen in ausführbare Modelle unerlässlich sind [CD+05]. Die schiere Anzahl einzuhaltender Regeln (derzeitig 200 bei DaimlerChrysler) verlangt jedoch deren automatisierte Prüfung am Modell. Hierfür existieren bereits Werkzeuge, die auf Basis von MATLAB M-Scripts Regelverletzungen aufdecken, wie z.B. der *Simulink Model Advisor* [MW06] oder *MINT* [Ric06]. Bei der Suche nach Modellierungsmustern bzw. Regelverletzungen, die auf zu vermeidenden Mustern basieren, zeigen diese Skripte aber deutliche Nachteile. Die Programmierung der Mustersuche ist aufwändig; die Skripte sind nur von Experten zu verstehen und schwer zu warten (siehe etwa Abb. 7 am Ende des Beitrags).

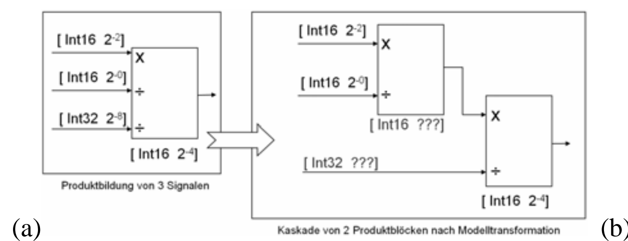

Abbildung 1: Problem bei der automatisierten Transformation eines Product-Blocks

Am folgenden Beispiel soll erläutert werden, welche Probleme bei der Anwendung von Modellierungsrichtlinien entstehen, wenn Modelle entsprechend dieser Regeln automatisiert (a) analysiert und (b) modifiziert werden sollen. Arithmetische Operationen, wie z.B. die Produktbildung zweier oder mehrerer Operanden, werden in Simulink mit Hilfe funktionaler Blockschaltbilder realisiert. Abb. 1 (links) zeigt einen so genannten Product-Block, mit drei Eingängen, deren Festkomma-Datentypen (int16, int32) auf unterschiedliche Genauigkeiten skaliert sind (2^{-0}, 2^{-2}, 2^{-8}). Das Ergebnis der Produktbildung wird an den Blockausgang des Product-Blocks propagiert. Diese Art der Modellierung ist problematisch, wenn das Modell mittels eines Codegenerators in Festkomma-Code übersetzt werden soll. Für die Übersetzung eines Product-Blocks mit mehr als zwei Operanden werden Skalierungsinformationen für Zwischenergebnisse benötigt (siehe Abb. 1, rechts), die der Codegenerator nicht automatisiert berechnen kann und die in Abb. 1 (a) auch nicht vom Entwickler angegeben werden können. Daher werden Product-Blöcke mit mehr als zwei Signaleingängen für die Seriencodegenerierung verboten.

Es erscheint sinnvoll, Entwickler durch die automatisierte Aufdeckung und halbautomatische Elimination solcher Richtlinienverletzungen zu unterstützen. Für die Entwicklung entsprechender Überprüfungen eignen sich Werkzeuge, die auf Graphtransformationen basieren, wie sie etwa vom Open-Source-Projekt *Fujaba* [Fuja06] mit dem Metamodellierungs-Plugin *MOFLON* [MOF06] angeboten werden. *Fujaba/MOFLON* erlaubt die visuelle Spezifikation verbotener Muster sowie deren Ersetzung durch empfohlene Entwurfsmuster auf hohem Abstraktionsniveau bei gleichzeitiger Unterstützung der OMG-Metamodellierungs-Standards zur modellbasierten Softwareentwicklung (MOF 2.0).

Im Folgenden werden wir zeigen, wie aufbauend auf einem Prototypen [GM+06] im Rahmen des MATE-Projekts die oben skizzierten Ideen exemplarisch umgesetzt wurden. Es wurde dafür zunächst in Kooperation mit dem artverwandten MESA-Projekt [FHR06] ein Metamodell für alle Modellierungselemente von MATLAB Simulink/Stateflow als MOF 2.0 (und damit auch als UML 2.0) Klassendiagramm entwickelt. Zur Laufzeit wird der Zugang zu Instanzen dieses Metamodells auf zwei verschiedene Arten unterstützt: (1) mittels Lese- und Schreibzugriffen über einen *Adapter* direkt auf die in Matlab Simulink/Stateflow gespeicherten Modelle und (2) durch Export der zu untersuchenden Modelle in ein externes *Repository* (mit anschließendem Re-Import) mit einer entsprechenden Schnittstelle für Lese- und Schreibzugriffe (siehe auch Abschnitt 5). Aufbauend auf dem Metamodell werden gemäß der Modellierungsrichtlinien zu suchende Muster mit Hilfe von *Analyseregeln* beschrieben, die dann mittels eines Musterinterferenzmechanismus von Fujaba [NS+02] in einem MATLAB Simulink- oder Stateflow-Modell identifiziert werden. Nachfolgend können dann *Transformationsregeln* in Form von Story-Diagrammen (siehe Abschnitt 3 und [NNZ00]) dazu genutzt werden, die ggf. vorhandenen Regelverletzungen durch Modelltransformationen zu korrigieren.

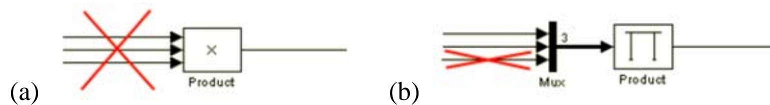

Abbildung 2: Betrachtete Fälle zur Ableitung von Analyseregeln

Ein erster notwendiger Schritt, um die in Abb. 1 angedeutete automatische Transformation zu ermöglichen, ist die Identifikation von verbotenen Situationen (siehe Abb. 2) mittels Analyseregeln. So darf, wie in Abb. 2 (a) angedeutet, kein Product-Block mit mehr als zwei Eingängen vorhanden sein oder, wie in Abb. 2 (b) gezeigt, kein vektorwertiger Eingang mit mehr als zwei Elementen existieren.

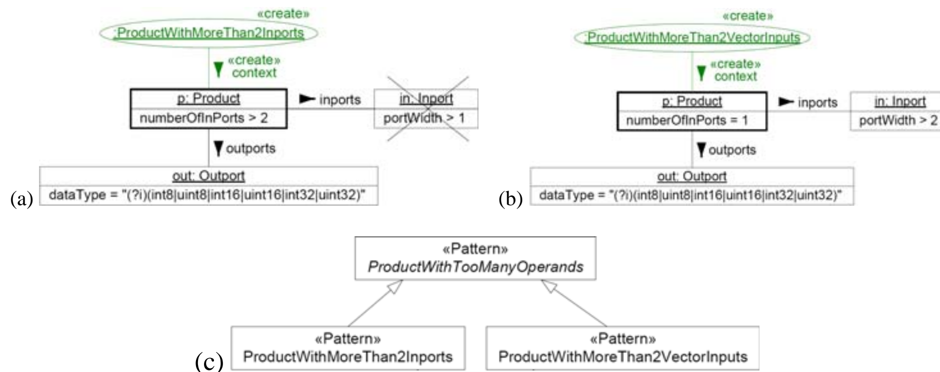

Abbildung 3: Analyseregeln für Produktblöcke mit zu vielen Eingängen

In Abb. 3 (a) und (b) sind zwei verschiedene Regeln dargestellt, die zur Überprüfung der Modellierungsrichtlinie aus Abb. 1 auf Basis des erstellten Metamodells dienen. In der Regel aus Abb. 3 (a) wird dabei mittels des Metamodell-Attributs numberOfInPorts spezifiziert, dass ein Produktblock (p:Product) mehr als zwei Eingänge hat, und dass der Da-

tentyp des Ausgangsports einem der relevanten Festkommaformate entspricht. Wird eine solche Situation gefunden, so wurde eine Verletzung der Modellierungsrichtlinie aus Abb. 2 (a) erkannt und die Situation wird durch eine entsprechende Annotation vom Typ ProductWithMoreThan2Inports am Product-Block kenntlich gemacht. Um auszuschließen, dass die Erkennung und anschließende Problembehandlung fälschlicherweise auch auf Blöcken mit vektorwertigen Inports ausgeführt wird, ist dies durch einen negativen vektorwertigen Inport-Knoten (ausgekreuzt) ausgeschlossen. Die zweite Analyseregel in Abb. 3 (b), die den Fall aus Abb. 2 (b) betrachtet, verbietet einen vektorwertigen Inport mit mehr als 2 Elementen. Beide Fälle lassen sich, wie in Abb. 3 (c) beschrieben, zu einem allgemeineren Muster namens ProductWithTooManyOperands zusammenfassen. Diese Verallgemeinerung trifft immer dann auf einen betrachteten Modellausschnitt zu, wenn eine ihre Spezialisierungen zu dem betrachteten Ausschnitt passt.

3 Modelltransformation

Neben der Erkennung von Regelverletzungen können *unidirektionale Transformationsregeln* in Form von Story-Diagrammen dazu genutzt werden, an die Analyseregel anknüpfende Modelltransformationen zu realisieren. So kann die Identifikation der entsprechenden Instanzsituation im Modell durch die vorgestellten Regeln aus Abb. 3 als Ansatzpunkt dazu genutzt werden, den Fehler durch eine Modelltransformation zu beheben. Dazu werden einfach weitere Elemente, die für eine Transformation notwendig sind, wie z.B. die über die Links verbundenen Blöcke, entsprechend eingeblendet.

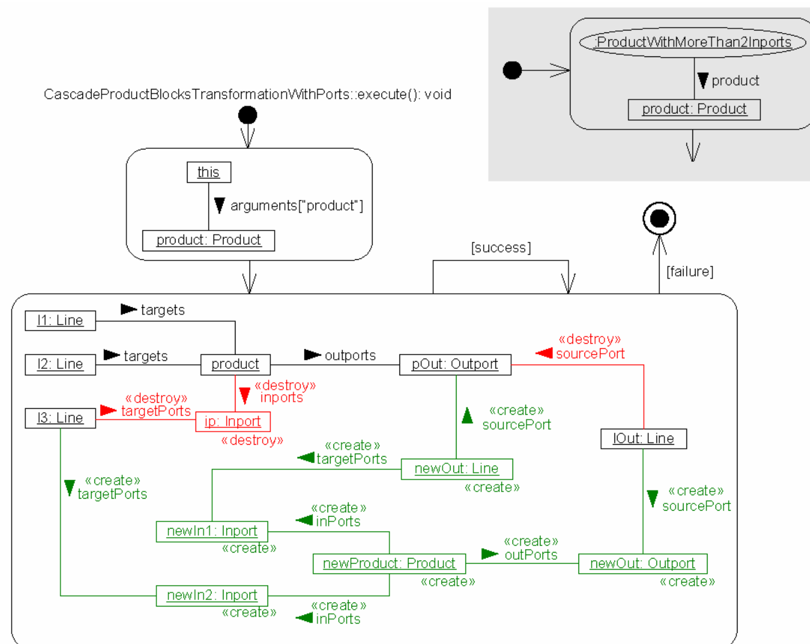

Abbildung 4: Transformationsregel für Blöcke mit mehr als zwei Eingängen

Die entsprechende Transformationsregel in Abb. 4 setzt auf der Erkennung der Situation auf und beschreibt, welche Veränderungen im Modell notwendig sind, damit die Modellierungsrichtlinie eingehalten wird. Dabei wird in der ersten Aktivität des dazu verwendeten erweiterten UML-Aktivitätsdiagramms die zuvor gefundene Situation durch Zugriff über eine qualifizierte Assoziation gebunden. Wie rechts oben in Abb. 4 angedeutet, soll dieses Binden durch entsprechend an der Musterdefinition orientierte Visualisierung zukünftig noch eingängiger gestaltet werden, indem man die in den Analyseregeln aus Abb. 3 verwendeten Annotationen direkt einblendet. Sobald die entsprechenden Elemente gebunden sind, wird im Aktivitätsdiagramm die zweite Aktivität angestoßen. Hier erlauben die in die Aktivität visuell eingebetteten Story Pattern die benötigte Transformation mittels zu löschender («destroy») oder zu erzeugender («create») Modellelemente zu spezifizieren. Durch eine Schleife im Aktivitätsdiagramm wird dabei beschrieben, dass dieser atomare Transformationschritt solange angewendet wird, bis keine Anwendungsstelle mehr gefunden werden kann.

An den gezeigten Beispielregeln kann man erkennen, wie die für die Verletzung von Modellierungsrichtlinien charakteristischen Situationen systematisch durch ausschnittartige Instanzsituationen des Metamodells und zusätzliche Bedingungen in Form der vorgestellten Regeln spezifiziert werden können.

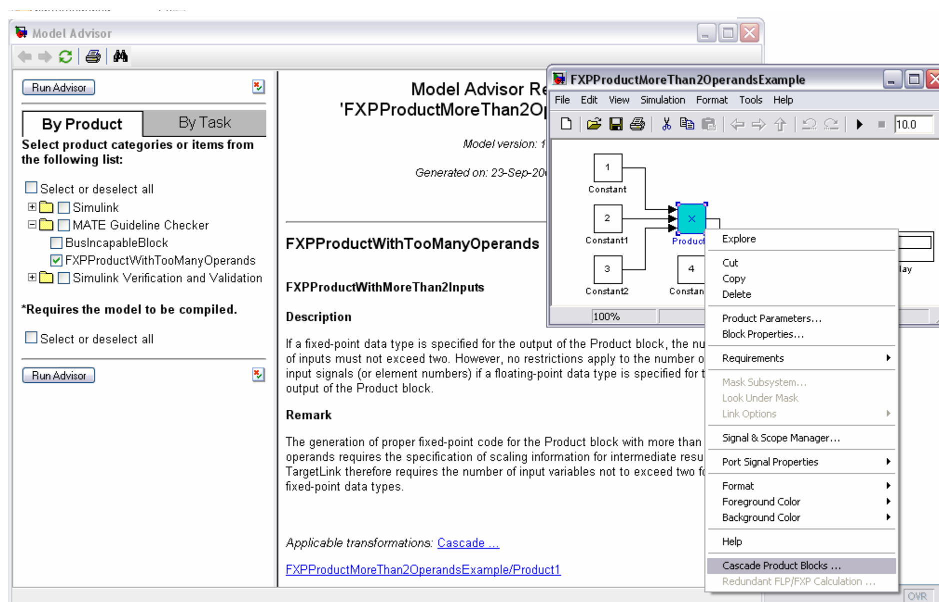

Abbildung 5: Analyseregeln und Transformationsregeln im ModelAdvisor

In Abb. 5 ist die Integration der Analyse- und Transformationsregeln in den Simulink ModelAdvisor angedeutet. Die Analyseregeln werden als normale Konsistenzregeln angezeigt und können entsprechend vom ModelAdvisor angestoßen werden. In diesem Fall ist dies die Regel ProductWithTooManyOperands.

Ist eine Situation gefunden worden, bei der die Regel verletzt wird, so wird durch den ModelAdvisor die entsprechende Stelle per Link zugänglich gemacht und ein die entsprechende Regel erläuternder Text angezeigt. Gibt es anwendbare Transformationsregeln, die zu einer Korrektur der Situation führen würden, so werden diese nun ebenfalls angezeigt. So wird in unserem Beispiel die Regel CascadeProductBlocksTransformationWithPorts dargestellt. Daneben ist, wie in Abb. 5 gezeigt, aber auch eine Auswahl derselben Regel für die selektierte Situation aus dem Kontextmenü in Matlab/Simulink möglich.

4 MATE Architektur

Die in Abb. 6 dargestellte Architektur von MATE bildet das Grundgerüst für alle skizzierten Anwendungsszenarien. Die Basis des Gesamtsystems bildet das Modellierungswerkzeug MATLAB Simulink/Stateflow mit der Erweiterung um den ModelAdvisor, der bereits ein interaktives Rahmenwerk für die Verwaltung und Ausführung von Analyseregeln sowie die Betrachtung von Analyseergebnissen auf *einem* Modell bereitstellt. Damit integriert wurde ein Visualisierungswerkzeug, das auf einigen im ToolNet-Projekt [ADS02] entwickelten Bausteinen zur Kommunikation mit MATLAB Simulink und Stateflow aufsetzt und beispielsweise die Markierung beliebiger Modellierungselemente drastisch vereinfacht. Dabei wird vor allem die gleichzeitige Betrachtung *mehrerer* Modellversionen und deren Unterschiede unterstützt.

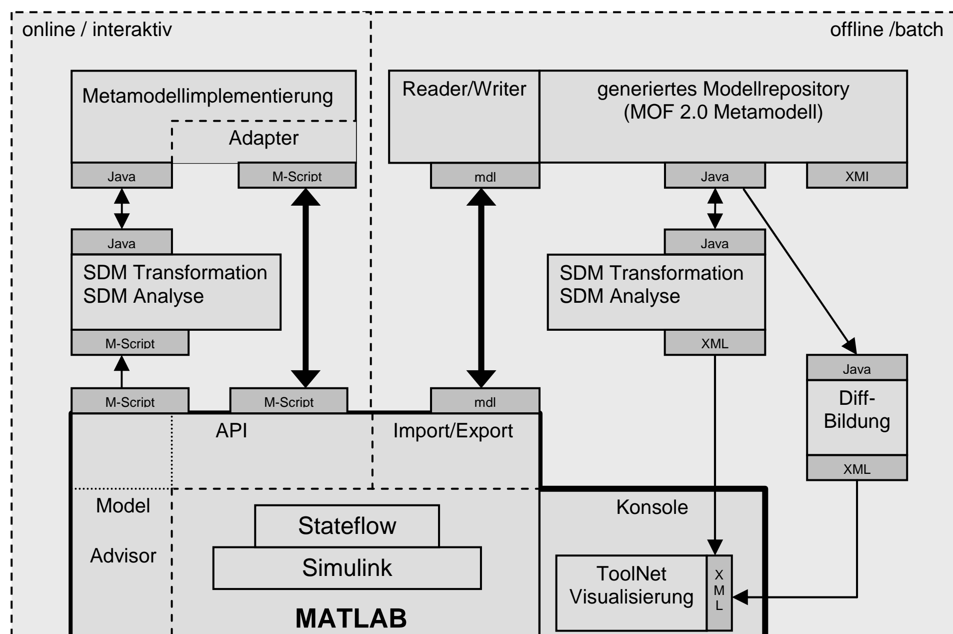

Abbildung 6: MATE-Architektur (vereinfacht)

Auf der beschriebenen Basisschicht setzen nun zwei weitgehend unabhängige Teilsysteme auf, die beide als Kernstück von Fujaba generierten Java-Code für Modell-Analysen und -Transformationen enthalten. Als Basis wird hierfür das in den Abschnitten 2 und 3 vorgestellte *S*tory *D*riven *M*odeling (SDM) eingesetzt.

Die *Online*-Lösung unterstützt zunächst die durch den ModelAdvisor gesteuerte interaktive Analyse von Modellen. Dabei ruft der ModelAdvisor *M-Script-Fragmente*[†] auf, die ihrerseits Java-Methoden aktivieren. Diese Methoden unterstützen die Identifikation empfohlener Design- oder verbotener Anti-Pattern in den untersuchten Modellen sowie die Ersetzung gefundener Anti-Pattern durch geeignete Design-Pattern. Sie kommunizieren dabei über proprietäre Schnittstellen mit einem Adapter, der alle auf Modellen denkbaren Lese- und Schreiboperationen unterstützt. Dieser Adapter greift wiederum über die M-Script-Schnittstelle auf die in MATLAB Simulink/Stateflow verwalteten Modelle zu. Die Analyseergebnisse werden auf umgekehrtem Weg zurückgegeben und im ModelAdvisor angezeigt. Auf demselben Weg ist es möglich, Transformationen auf durch die Analyse als fehlerhaft gekennzeichneten oder anderweitig selektierten Modellstellen interaktiv durchzuführen.

Die *Offline*-Lösung geht bei der Analyse und Transformation von Simulink/Stateflow-Modellen anders vor, da hier nicht die interaktive Manipulation von Simulink/Stateflow-Modellen, sondern aufwändige Analysen im Mittelpunkt stehen, die den Aufbau zusätzlicher Index-Strukturen erfordern. Deshalb werden zunächst Simulink- und Stateflow-Modelle in ihrem propietären mdl-Format exportiert und in ein eigenständiges Modell-Repository importiert. Dieses Repository verfügt über eine Java-API, die den von Sun entwickelten Standard JMI unterstützt. Zur Datenhaltung kann deshalb entweder das Metadaten-Repository MDR von Sun oder bevorzugt eine von den Projektpartnern selbst entwickelte und deutlich effizientere Lösung verwendet werden, die alle Modellierungsdaten im Hauptspeicher hält. Letztere wird aus dem in Abschnitt 2 erwähnten MATLAB Simulink/Stateflow-Metamodell mit Hilfe des Fujaba-Plugins MOFLON generiert.

Auf dem generierten Repository werden die zur Verfügung stehenden Analyse- und Transformationsmethoden ausgeführt, die einerseits die im Repository abgelegten Modelle verändern und andererseits Analyseberichte in Form von XML-Dateien erzeugen können. Die veränderten Modelle können wiederum als mdl-Dateien in die MATLAB Simulink/Stateflow Umgebung eingelesen werden. Die XML-Analyseberichte werden von der bereits oben erwähnten ToolNet-Visualisierung weiterverarbeitet. Sie öffnet sowohl das analysierte Modell als auch das daraus hervorgegangene korrigierte Modell und zeigt jeweils zueinander korrespondierende fehlerhafte Stellen im Ursprungsmodell und korrigierte Stellen im neuen Modell an. Die zur Analyse und Transformation eingesetzten Java-Methoden werden bei der Offline-Lösung ebenfalls visuell spezifiziert und mit Hilfe von Fujaba/MOFLON in Java-Code übersetzt. Ähnlich funktioniert die Berechnung von Modelldifferenzen, die zwei im Repository abgelegte Modelle miteinander vergleicht und deren Unterschiede mit dem selben Visualisierungsmechanismus wie die Analysen und Transformationen darstellt.

[†] M-Script ist die für die Manipulation von Simulink- und Stateflow-Modellen angebotene Script-Sprache.

6 Verwandte Arbeiten

Es gibt bereits eine Handvoll von Ansätzen wie den Simulink Model Advisor [MW06] oder MINT [Ric06], die meist auf der Basis von MATLAB M-Scripts Regelverletzungen in Simulink- oder Stateflow-Modellen aufdecken. Wie an Abb. 7 ersichtlich (im Anhang der Veröffentlichung), ist die „Programmierung" solcher Regeln jedoch erheblich komplexer und aufwändiger als die im MATE-Projekt realisierte Spezifikation von Analyseregeln und anschließende Generierung der ausführbaren Regeln. Dies liegt zum einen daran, dass die von uns verwendete Adapterschicht immer wiederkehrende Zugriffscodeblöcke geeignet kapselt und diese somit wiederverwendet werden. Darüber hinaus wird vor allem in den Analyse- und Transformationsregeln auf einem höheren Abstraktionsniveau spezifiziert, so dass aufwändige Navigationsoperationen auf dem Metamodell in den Regeln wesentlich kompakter beschreibbar sind.

Des Weiteren existieren Arbeiten im Rahmen des MESA-Projekts [FHR06], die ebenfalls auf einem höheren Abstraktionsniveau Modellierungsrichtlinien spezifizieren und daraus ausführbaren Code generieren. Ausgangspunkt ist auch dort das von uns mitverwendete MATLAB Simulink/Stateflow-Metamodell; allerdings wird anstelle von visuellen regelbasierten Spezifikationstechniken in MESA die textuelle logikbasierte Object Constraint Language (OCL) der OMG eingesetzt. Ein umfassender Vergleich von Fujaba SDM-Diagrammen einerseits und OCL Constraints andererseits in punkto Ausdrucksstärke, Erstellungsaufwand, Lesbarkeit und industrieller Akzeptanz liegt bislang noch nicht vor. Klar ist allerdings, dass SDM-Diagramme direkt die Spezifikation von Modelltransformationen unterstützen, während bei OCL zusätzliche Erweiterungen zu einer Transformationssprache benötigt werden. Darüber hinaus eignen sich SDM-Diagramme eher zur Beschreibung komplexer Entwurfsmuster, während OCL eher auf die Beschreibung logischer Formeln mit Konjunktionen und Implikationen zugeschnitten ist. So erscheint uns auf Dauer die Kombination von SDM und OCL sinnvoll, wie sie im Rahmen der Metamodellierungsumgebung Fujaba/MOFLON bereits in Angriff genommen wurde.

Abschließend sei darauf hingewiesen, dass keine der verwandten Arbeiten den Bogen von rein batchorientierten Analyseregeln hin zu interaktiven Modelltransformationen spannt, sodass nur MATE neben der Erkennung von Richtlinienverletzungen auch deren Behebung entweder im Batchbetrieb oder mit komplexen Editierkommandos unterstützt.

7 Zusammenfassung

Das Projekt MATE – eine Kooperation von DaimlerChrysler mit vier Universitäten und der Firma Model Engineering Solutions – stellt eine Architektur, Vorgehensweisen und Werkzeuge zur Verfügung, die die Lösung praxisrelevanter Probleme der modellbasierten Entwicklung mit MATLAB Simulink/Stateflow ermöglicht. Die Realisierung des Ansatzes wurde am Beispiel der automatisierten Überprüfung und Anwendung von Modellierungsrichtlinien und Modelltransformationen beschrieben, die mit Hilfe von Fujaba und dem dazugehörigen Metamodellierungs-Plugin MOFLON auf hohem Abstraktionsniveau spezifiziert und in ausführbaren Java-Code übersetzt werden.

Durch Anwendung weiterer in *Fujaba* verfügbarer Konzepte wie *bidirektionale Konsistenzregeln* [GW06] können auf Basis der im ToolNet-Projekt [ADS02] entwickelten Infrastruktur zur Integration von Werkzeugen auch darüber hinaus benötigte Aspekte wie die (halb-)automatische Überwachung von Konsistenzbeziehungen und Propagation von Änderungen zwischen MATLAB Simulink/Stateflow-Modellen einerseits und in DOORS verwalteten Anforderungen umgesetzt werden.

Darüber hinaus wurde die MATE-Architektur um SiDiff [KWN05, SiDiff], ein generisches Werkzeug zur Berechnung symmetrischer Differenzen zwischen zwei graphartig strukturierten Dokumenten erweitert. SiDiff selbst ist zu diesem Zweck auf den Vergleich von MATLAB Simulink-Modellen so konfiguriert worden, dass nur wesentliche Änderungen hervorgehoben werden und vor allem selbst umbenannte oder verschobene Modellierungselemente auf Basis entsprechender Heuristiken einander zugeordnet werden können. Die gefundenen Differenzen werden gegenwärtig durch paarweises Hervorheben korrespondierender Elemente in Matlab/Simulink visualisiert. Eine Erweiterung von SiDiff auf Stateflow-Modelle wird in Kürze zur Verfügung stehen.

Literatur

[ADS02] Altheide, F.; Dörr, H.; Schürr, A.: Requirements to a Framework for sustainable Integration of System Development Tools, in: *Proc. of 3rd European Systems Engineering Conference (EuSEC'02)*, Toulouse: AFIS PC Chairs (2002), 53-57

[CD+05] Conrad, M.; Dörr, H.; Fey, I.; Pohlheim, H.; Stürmer, I.: Guidelines und Reviews in der modellbasierten Entwicklung von Steuergeräte-Software, in: *Simulation und Test in der Funktions- und Softwareentwicklung für die Automobilelektronik*, Expert-Verlag (2005)

[FHR06] Farkas T.; Hein, Ch.; Ritter, T.: Automatic Evaluation of Modelling Rules and Design, in: *Second Workshop "From code centric to model centric software engineering: Practices, Implications and ROI"*. Bilbao, Spain (2006)

[GW06] Giese, H.; R. Wagner: Incremental Model Synchronization with Triple Graph Grammars, in *Proc. of 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, Genoa, Italy, LNCS 4199, Springer Verlag (2006), 543-557

[KWN05] Kelter, U.; Wehren, J.; Niere, J.: A Generic Difference Algorithm for UML Models, in: *Proceedings of the SE 2005*, Essen, Germany (2005)

[Fuja06] *Fujaba*, http://www.fujaba.de and ./projects/reengineering/index.html (2006)

[GM+06] Giese, H.; Meyer, M.; Wagner, R.: A Prototype for Guideline Checking and Model Transformations for MATLAB/Simulink, in: *Proc. of the Fujaba Days* (2006)

[MW06] The MathWorks: http://www.mathworks.com/products (2006)

[NNZ00] Nickel, U.; Niere, J.;. Zündorf, A.: Tool demonstration: The FUJABA environment, in *Proc. of the 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, ACM Press (2000), 742-745

[NS+02] Niere, J.; Schäfer, W.; Wadsack, J.; Wendehals, L.; Welsh, J.: Towards Pattern-Based Design Recovery, in *Proc. of the 24th International Conference on Software Engineering (ICSE)*, Orlando, Florida, USA, ACM Press (2002), 338-348

[Ric06] Ricardo, Inc.: *MINT*, http:/www.ricardo.com/mint (2006)

[SiDiff] Universität Siegen: SiDiff Difference Tool Set; www.sidiff.org (2006)

[Tel06] Telelogic, Inc.: *DOORS*, http://www.telelogic.com (2006)

[MOF06] Technische Universität Darmstadt: *MOFLON*, http://www.moflon.org (2006)

```
  1  function [ok, test_data] = mint_tl_0009(cmd_s, scope_list_st)
  2  % MINT_TL_0009 --  Limitations with regard to Operand Numbers for the Product Block
  3  %

...

 10  %
 11  % Copyright 2006 DaimlerChrysler AG
 12  % (Written by I. Stuermer)
 13  %

...

 23  % Initialize the return variable.
 24  ok=1;
 25  % Registration data.
 26  if strcmp(cmd_s,'register')
 27  test_data.name = 'TL_0009 - Limitations with regard to Operand
                      Numbers for the Product Block';
 28    %test_data.features = {'mint_dcag'};
 29    test_data.number = 10009;
 30    test_data.scope = 'subsystem';
 31    test_data.category = 'Warning' ;
 32    test_data.ispseudo = 0 ;
 33    test_data.generated_scope = '' ;
 34    test_data.data_gen = { ...
 35      } ;
 36    test_data.data_req = { ...
 37        'subsys_handle'...
 38      } ;
 39
 40    return ; %%%%%%% RETURN %%%%%%%
 41  end

 42  % Ticker update.
 43  ok=mint_tdata_run_progress;
 44  if(~ok)
 45    return;
 46  end

 47  % Unpack subsystem handles
 48  subsys_handle = mint_tdata_data_get(scope_list_st,...
 49    'subsys_handle');
 50  subsys_handle = [subsys_handle{:}];

 51  top_h = mint_get_handle(mint_tdata_top_get);
 52  model_name=get_param(top_h,'Name');

 53  % Create list of erroneous product blocks inside controller
 54  f_block_h = [];

 55  % create list of all product block with fixed-point data types
 56  all_block_h= [] ;

 57  % We want to check the CompiledPortDataType of the Product outport
 58  % In order to do so, we need to set the model into compile mode.
 59  % Before, term compilation if model is already in compile mode
 60  try
 61   eval(sprintf('%s([],[],[],''term'')',model_name));
 62  end

 63  try
 64    %switch to compile mode
 65     eval(sprintf('%s([],[],[],''compile'')',model_name));
 66
 67  catch
 68     disp('TL_0009 - Unable to set model into compile mode');
 69     errstr = strrep(lasterr, 'Error using ==> eval', '');
 70     stern = '*********************************************************';
 71     fstr = sprintf('\n%s\n%s\n\n%s\n\n', stern, errstr, stern);
 72     disp(fstr);
 73  end

 74  % get all product blocks
 75  block_h = mint_find_system(top_h,...
 76               'BlockType','Product');
 77  all_block_h=[all_block_h;block_h];

 78  %loop over all product blocks
 79  for k=1:length(all_block_h)
 80
 81    %get port handles of actual product block
 82    porth= get_param(all_block_h(k),'PortHandles');
 83
 84    % check whether its outport data type is an integer (fixed-point)
 85    % data type
 86    outdata= get_param(porth.Outport,'CompiledPortDataType');

 87    % prepare regular expression for query
 88    regexp= '[int8|int16|int32|fix]';
 89    res=regexpi(outdata, regexp, 'match');
 90
 91    %get the name of the actual block
 92    block_name=get_param(all_block_h(k),'Name');
 93
 94    % if res is not empty, we found a product block whose outport data type
 95    % is (scaled) integer
 96    if ~isempty(res)
 97
 98      %disp(sprintf(' Output data type of block %s is %s', block_name, outdata));

 99
100      % CHECK WHETHER PRODUCT BLOCK HAS MORE THAN TWO INPUTS
101      if (length(porth.Inport) > 2)
102       f_block_h = [f_block_h;all_block_h(k)];
103      else
104        % So we got only two inports!
105
106        % CHECK WHETHER PRODUCT BLOCK IS DRIVEN BY A VECTOR
107        % We have to consider two cases
108        % (1) one inport with vector > 2
109        % (2) two inports   (a) first inport with vector >=2, or
110        %           (b) second inport with vector >= 2
111
112        % CASE 1
113        if (length(porth.Inport)==1)
114          indata= get_param(porth.Inport(1),'CompiledPortDimensions');
115          vector_size= indata(2);
116          if ((vector_size) > 2)
117           f_block_h = [f_block_h;all_block_h(k)];
118            %disp(sprintf(' Input dimension of block %s inport 1 is %s',
                           block_name, vector_size));
119          end
120        else
121          % CASE 2
122          if (length(porth.Inport)==2)
123            indata= get_param(porth.Inport(1),'CompiledPortDimensions');
124            vector_size= indata(2);
125            % CASE 2 a
126            if ((vector_size) > 1)
127             f_block_h = [f_block_h;all_block_h(k)];
128              %disp(sprintf(' Input dimension of block %s inport 1 is %s',
                             block_name, vector_size));
129            else
130              % CASE 2 b
131              indata= get_param(porth.Inport(2),'CompiledPortDimensions');
132              vector_size= indata(2);
133              if ((vector_size) > 1)
134               f_block_h = [f_block_h;all_block_h(k)];
135               %   disp(sprintf(' Input dimension of block %s inport 1 is %s',
                                 block_name, vector_size));
136              end
137            end
138          end
139        end
140      end
141    end
142  end

143  try
144   eval(sprintf('%s([],[],[],''term'')',model_name));
145  end

146  f_block_parent_h = mint_parent_h(f_block_h);

147  % Find all relevant subsystems
148  [dummy,subsys_i,dummy] = intersect(subsys_handle,f_block_parent_h);

149  % loop over subsystems
150  for j = subsys_i
151    % Check that the intersection is empty
152    blocks_i = find(subsys_handle(j) == f_block_parent_h);
153
154    if (isempty(blocks_i))
155      % set pass to the subsystem
156      subsys_name_s_c = mint_getfullname_c(subsys_handle(j));
157      mint_tdata_new_passmarkers(subsys_name_s_c, ...
158        scope_list_st, ...
159        1);
160    else
161      subsys_name_s_c = mint_getfullname_c(subsys_handle(j));
162      name_all_s = 'Limitations with regard to Operand Numbers for the Product Block';
163      info_s{1} = 'A product block with a fixed-point data type specified for the
output must comply with the following rules:';
164      info_s{2} = ' - the number of inports must not exceed two';
165      info_s{3} = ' - when driven by a vector the number of signals within this vector
must not exceed two.';
166      block_name_s_c = mint_get_param_c(f_block_h(blocks_i),'Name');
167      [sel_text_s_c, sel_handles_c] = ...
168        mint_make_selection2(name_all_s, ...
169        block_name_s_c, f_block_h(blocks_i));
170      mint_tdata_new_message(subsys_name_s_c, ...
171        mint_tdata_desc_get(scope_list_st, j), ...
172        sel_text_s_c, ...
173        sel_handles_c, ...
174        info_s);
175    end
176  end
```

Abbildung 7: Manuell erstelltes M-Skript mit derselben Funktionalität wie die Analyseregeln aus Abb. 4