UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik, Informatik und Mathematik
Institut für Informatik

Seminar: Softwareengineering für softwareintensieve Systeme

Thema

Grundlagen der Codegenerierung & WCET

von
Andreas Seibel

23. August 2006

Inhaltsverzeichnis

1 Motivation

Software gewinnt einen immer höheren Stellenwert in technologisch fortgeschrittenen Gesellschaften und ist somit zu einer Schlüsselrolle für Produkte und Dienste in Wirtschaft und Gesellschaft geworden. Es ist ein enormes Wachstum im Bereich der eingebetteten Echtzeitsysteme, im Vergleich zu den herkömmlichen Computersystemen aus dem Home- / Office-Bereich, zu verzeichnen. Menschen in diesen technologisch fortgeschrittenen Gesellschaften werden mehr und mehr, auch wenn indirekt, von der Interaktion mit Softwaresystemen im täglichen Leben abhängen, wie z.B. im Transportwesen, Bankwesen, der Medizin, der Telekommunikation und dem Servicewesen.

Aber die intensive Verbreitung von Softwaresystemen ist nicht das einzige Phänomen der heutigen Zeit. Softwaresysteme selbst weisen einen stetigen Wachstum ihrer Komplexität vor [10]. So veranlasst die ständige Leistungssteigerung der Computersysteme gleichzeitig das Potential der Softwareentwicklung, in dem die Software um neue Besonderheiten erweitert wird die zuvor auf Grund der fehlenden Rechenkapazitäten nicht implementiert wurden. Aber auch die ständig steigende Abstraktion der Programmierparadigmen führt indirekt und nachhaltig zu einer Steigerung der Softwarekomplexität. Paradigmen mit höherem Abstraktionsniveau verschieben den Konzentrations-Fokus auf logische Bereiche des Problems und schaffen dadurch mehr Freiraum, um auf das eigentliche Problem einzugehen. Die Beschreibung des Problems wird durch höhere Abstraktionsebenen zwar vereinfacht, das resultierende Programm wird allerdings immer noch in Hochsprachen wir Java oder C++ umgesetzt was nicht zu einer Vereinfachung der Implementierung führt, im Gegenteil.

Im Bezug auf effiziente Softwareentwicklung, ist modellbasierte Softwareentwicklung (MDD) ein guter Ansatz, um die Entwicklungszeit und die damit verbundenen Entwicklungskosten drastisch zu mindern. Hauptverantwortlich für die drastische Minderung der Entwicklungszeit ist, vor allem, der automatische Prozess, welcher Modelle in zielplattform-spezifischen lauffähigen Quellcode übersetzt. Dieser Prozess wird auch Codegenerierung genannt. Viele Werkzeuge – z. B. Together[1], Magic Draw[2], Poseidon[3], AndroMDA[4] und Fujaba[5] – bieten mittlerweile die Modellierung von Struktur sowie Verhalten an, womit bereits komplette Softwaresysteme als Modell beschrieben werden können, aus denen automatisch Quellcode erzeugt werden kann. Code-Generatoren verbinden somit die Entwicklungsphasen des Softwareentwurfs und der Implementierung. Dies lässt sich besonders gut in dem Vergleich der beiden Softwareentwicklungs-Prozessmodelle V und Y zeigen (siehe Abbildung 1).

Beide Diagramme sind in x-Richtung auf der Zeit abgetragen. Die y-Achse beschreibt die einzelnen Phasen des Entwicklungsprozesses. Es wird zwischen Entwick-

[1] http://www.borland.com/de/products/together/index.html
[2] http://www.magicdraw.com
[3] http://www.gentleware.com
[4] http://www.andromda.org
[5] http://www.fujaba.de

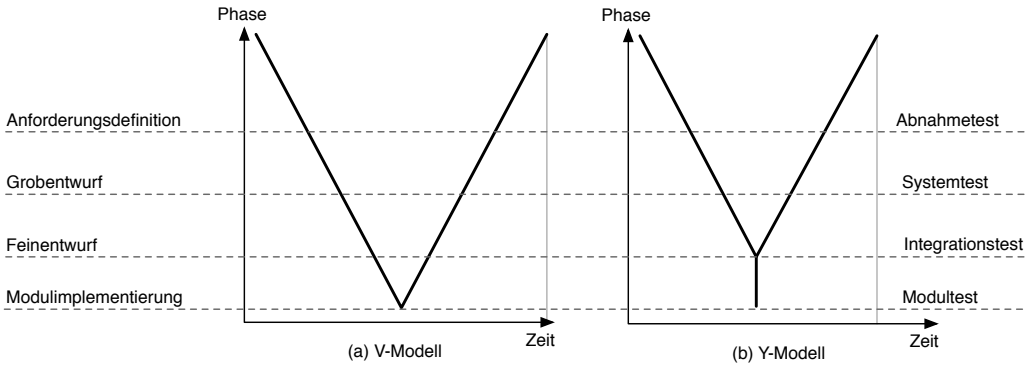

Abbildung 1: V- und Y-Prozessmodell im Vergleich

lungsphasen und Testphasen unterschieden die zum Einen auf der linken Seite der Kurve (negative Steigung) und der rechten Seite der Kurve (positive Steigung) abgetragen sind. Die Geraden mit negativer Steigung beschreiben die Entwicklungsphasen und die Geraden mit positiver Steigung die Testphasen, welche eine geringere Rolle in dieser Arbeit, bezüglich qualifiziertem Code bzw. qualifizierten Code-Generatoren, spielen (siehe Abschnitt 4.1). Bei dem V-Modell vergeht in jeder der Phasen eine gewisse Zeit. Interessant für diese Arbeit ist die Entwicklungsphase zwischen dem Feinentwurf und der Modulimplementierung. Dieser Prozess wird bei dem V-Modell noch mit einem zeitlichen Aufwand, der durch manuelle Implementierung hervorgerufen wird, versehen. Dieser zeitliche Aufwand kann aber durch eine automatische Codegenerierung gegen Null streben. Dadurch werden zum Einen durch automatische Codegenerierung die Phase zwischen Feinentwurf und Modulimplementierung, und zum Anderen die Phase zwischen Modultest und Integrationstest durch qualifizierten Code bzw. qualifiziertem Code-Generator, die Zeit gegen Null streben lassen . Das resultierende Modell wird auch als Y-Modell bezeichnet (rechte Seite in Abbildung 1).

Dadurch das Code-Generatoren i.d.R. ein deterministisches Verhalten aufweisen, existiert zu jedem möglichen Zustand des Modells auch nur genau ein eindeutiger Zustand des Quellcodes. Dies fördert die Reproduzierbarkeit des Codes und vermindert die Mehrdeutigkeit die, i.d.R. durch manuelle Implementierung auf Grund subjektivem und nicht eindeutigem Wissen des Programmierers, entsteht. Ein weiterer Vorteil, die Code-Generatoren gegenüber von Programmierern haben, ist die geringere Fehlerrate. Programmierer produzieren nicht selten Fehler, welche keinem Determinismus folgen was systematisches vermeiden der Fehler, im Grunde, unmöglich macht. Auch Code-Generatoren können keine Fehlerfreiheit garantieren aber ein, zumindest deterministischer, Code-Generator erzeugt, wenn überhaupt, deterministisch Fehler welche systematisch ausgeschlossen werden können und welche nach Korrektur des Code-Generators nicht erneut auftreten.

Da softwareintensive Systeme in naher Zukunft sich vermehrt auf eingebettete Systeme, und vor allem eingebettete Echtzeitsysteme, beziehen, wird in dieser

Domäne der Bedarf an modellbasierter Softwareentwicklung steigen. Auch hier gilt, dass der Markt die Bedingungen für Produkte zur Entwicklung von softwareintensiven Systemen, schafft. Stetig sinkende Entwicklungszyklen führen nachhaltig dazu, dass sich nur die Hersteller auf dem Markt etablieren, die es schaffen ein Produkt, zum Einen sicher und fehlerfrei zu produzieren, und zum Anderen der Konkurrenz immer einen Schritt voraus sind.

Ein wichtiger Aspekt bei eingebetteten Echtzeitsytemen sind die zeitlichen Restriktionen, die ein solches System ständig unterliegt. So gibt der eingebettete Kontext dieser Systeme, Teilen des Systems strikte maximale Laufzeiten (Deadlines) vor, die diese Teile bei ihren Berechnungen zu keinem Zeitpunkt überschreiten dürfen. Um dies zu garantieren werden Verfahren eingesetzt für die, zum Einen die maximale Laufzeit, oder auch *worst-case execution time* (WCET) genannt, eines Prozesses bestimmen, und zum Anderen Verfahren die gewährleisten, dass für die gegebenen WCET's ein *Scheduling* existiert, bei dem jeder Prozess seine gegebene Deadline nicht überschreitet.

1.1 Zielsetzungen

Das Ziel dieser Arbeit soll eine grundlegende Einführung in das Thema der Codegenerierung und der WCET sein. Bei der Codegenerierung kann allerdings kein Ansatz als solches beschrieben werden, welcher für Codegenerierung selbst stellvertretend ist, da ein solches Verfahren nicht existiert. Stattdessen werden Aspekte der Codegenerierung vermittelt, die sich auf zeitbehaftete sowie nicht-zeitbehaftete Systeme getrennt beziehen. Bezüglich der Codegenerierung zeitbehafteter Systeme wird eine kurze Einführung in den Bereich der eingebetteten Echtzeitsysteme gegeben. Parallel zur Erläuterung der spezifischen Aspekte der Codegenerierung für die unterschiedlichen Teilbereiche von Echtzeitsystemen, wird der jeweilige Teilbereich in den Abschnitten zu Beginn erläutert.

1.2 Struktur dieser Arbeit

Im Abschnitt 2 werden zuerst Grundlagen über Echtzeitsysteme, vor allem aber eingebettete Echtzeitsysteme erläutert. Anschließend wird in Abschnitt 3 die Notwendigkeit der WCET und Verfahren zur Bestimmung dieser erläutert. Abschnitt 4 beschreibt die Prinzipien der Codegenerierung und geht daraufhin auf Aspekte der Codegenerierung der Domäne von MDD für Echtzeitsysteme ein. Der letzte Abschnitt 5 gibt die obligatorische Zusammenfassung mit einem persönlichem Resümee zum Thema.

2 Echtzeitsysteme

Echtzeitsysteme sind immer mehr im Einsatz und bekommen eine immer größere Relevanz in der Software Engineering Disziplin. In vielen Domänen handelt es sich bei Echtzeitsystemen um eingebettete Echtzeitsysteme. Die Einbettung, aber auch ökonomische Umstände die Einsparungen bei den Hardwarekosten erzwingen, führen dabei zu Ressource-Restriktionen. Ressourcen beziehen sich dabei besonders auf die Leistung der Rechnerarchitektur (Prozessor) und den Speicher des Systems. Der Mangel beider Ressourcen wirkt sich zwangsläufig auf die Entwicklung von eingebetteten Echtzeitsystemen aus, da für die eingeschränkte Plattform Software entwickeln werden muss. Restriktiver Speicher führt dazu, dass die Software sparsam mit dieser Ressource verfahren muss, aber auch, dass zu keinem Zeitpunkt der Ausführung diese Ressource nicht überstrapaziert wird, was zu fehlerhaftem Verhalten des Gesamtsystems führen kann. Fehlende Leistung führt zu Optimierungen des auszuführenden Codes um gegebene Deadlines einhalten zu können.

Software eingebetteter Echtzeitsysteme kann in mehrere Prozesse partitioniert werden. Da i.d.R. nicht jeder Prozess dediziert von einem Prozessor ausgeführt werden kann, konkurrieren diese Prozesse untereinander um die Ausführung auf dem Prozessor. Welcher Prozess wann ausgeführt wird, hat das System anhand der Situation zu entscheiden, in der es sich befindet. Dabei existieren feste Zeitscheiben, in denen ein Prozess seine Ausführung abgeschlossen haben muss was auch Deadline genannt wird. Diese sind, vor allem bei eingebetteten Echtzeitsystemen, durch physikalische Gegebenheiten definiert. Folgendes Zitat von Don Gillies gibt diesbezüglich eine klare Definition von einem Echtzeitsystem.

> *„A real-time system is one in which the correctness of the computations not only depends upon the logical correctnes of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occured.“* [8]

Echtzeitsysteme, generell, können in zwei Klassifikationsstufen unterteilt werden. Zum Einen in die so genannten *harten* Echtzeitsysteme, und zum Anderen in die *weichen* Echtzeitsysteme. Diese Klassifizierungen unterscheiden sich in der Interpretation der logischen Korrektheit einer Berechnung bei Überschreitung einer Deadline. So wird bei diesen Systemen zwischen *harten* sowie *weichen* Deadlines unterschieden. Der Unterschied wird in Abbildung 2 grafisch dargestellt.

Bei *harten* Echtzeitsystemen (a) hängt vom Einhalten einer jeden Deadline eine große Verantwortung ab. So kann das Überschreiten einer harten Deadline im öffentlichen Transportwesen zu Unfällen führen, bei denen das Leben von Menschen gefährdet ist [2, 16].

Bei *weichen* Echtzeitsystemen (b) ist das Überschreiten der Deadline nicht von einer ultimativen Tragweite, wie bei *harten* Echtzeitsystemen, betroffen, stattdessen kann mit einem Komfortverlust gerechnet werden [5]. Bei Überschreitung einer

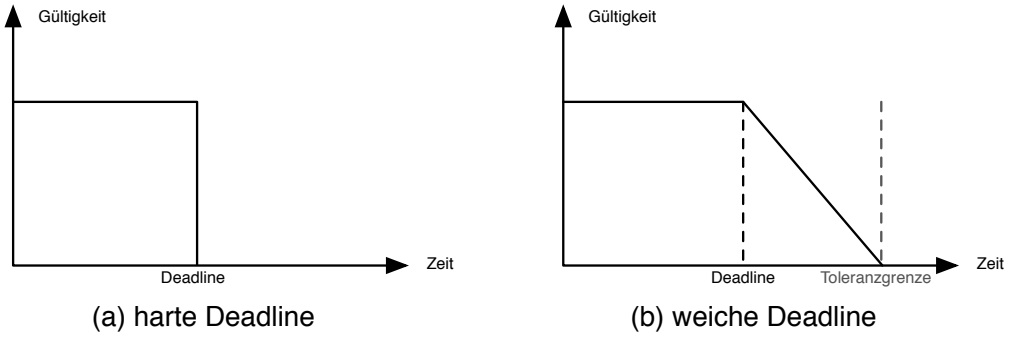

Abbildung 2: Vergleich zwischen *weichen* und *harten* Echtzeitsystemen

weichen Deadline existiert ein gewisser Toleranzbereich der die Deadline überschreiten darf, was nicht automatisch zur Ungültigkeit der Berechnung folgt. Ein Beispiel für *weiche* Echtzeitsysteme sind zeitkritische Multimedia-Anwendungen (Video/Audio-Produktion) oder im eingebetteten Bereich die Multimedia-Steuerung eines Automobils.

3 WCET

Um die angesprochenen Probleme aus dem vorangegangenen Kapitel zu vermeiden muss a-priori sichergestellt werden, dass jeder Prozess in jedem Fall die ihm gegebene Deadline nicht überschreitet. Dafür wird, unter anderem, vorausgesetzt, dass die maximale Ausführungszeit (*worst-case execution time* oder WCET) eines jeden Prozesses zu bestimmen ist. Liegt die WCET eines jeden Prozesses vor, so kann eine *Schedulability*-Analyse des Systems durchgeführt werden, um sicherzustellen, dass jeder Prozess durchgeführt werden kann, so dass die WCET des Prozesses nicht die ihm vorgegebene Deadline überschreitet.

Um die Bestimmbarkeit der WCET eines Prozesses zu gewährleisten muss das Programm, dass den Prozess ausführt, Vorhersagbar sein. Dies bedeutet, dass der Code der für den Prozess zuständig ist einen vollständigem Determinismus folgt. Dies wäre z. B. nicht gegeben wenn man die objektorientierte Hochsprache C++ und die in der *Standard Template Library* (STL) definierte HashMap Datenstrukturen verwendet. Diese verwendet, z.B. beim Ablegen von Elementen in die Datenstruktur, einen Nicht-Determinismus was die Analysierbarkeit des Codes bezüglich der WCET unmöglich macht. Echtzeitsysteme sollten daher nur durch Software repräsentiert werden, welche Sprachen oder Sprachdialekte verwenden, die ausschließlich deterministisch sind um so die Messbarkeit des System zu gewährleisten.

Die WCET eines Prozesses wird durch die WCET-Analyse bestimmt. Die WCET-Analyse bezieht sich immer auf eine Zielplattform auf der der Prozess ausgeführt werden soll. Ausserdem, wie zuvor exemplarisch gezeigt, ist dies auch abhängig von

der Implementierungs-Sprache die verwendet wurde. Nur für diese Konfiguration hat die analysierte WCET ihre Gültigkeit. Für die WCET-Analyse kann, ohne Einschränkungen, kein automatischer Ansatz für generischen Code existieren da sich eine allgemeine WCET-Analyse auf das *Halte-Problem* der Informatik reduzieren lässt [11]. Die WCET-Analyse lässt sich nur durch Restriktionen erreichen, indem die Programmiersprache der Eingabe auf ein analysierbares Minimum reduziert wird und indem zusätzliche Informationen des Programmflusses als zusätzliche Eingabe definiert werden.

Ein allgemeines Prinzip zur Bestimmung der WCET von ausführbarem Code, hat sich in der Vergangenheit bereits etabliert. Das Verfahren Bound-T[6] z.B., führt die WCET-Analyse auf bereits kompilierten Objekt-Code durch. Aus diesem Code wird die Strukturinformation, der so genannte *control flow graph* (CFG) [19] extrahiert. Dieser beschreibt den Fluss des Codes. Für die einzelnen Knoten (Blöcke) des CFG wird die WCET bestimmt. Mit diesen Informationen kann der schlechteste Pfad durch den CFG bestimmt werden um die WCET des Codes zu bestimmen. Bound-T behandelt allerdings nur einfache Prozessorarchitekturen, welche über kein *caching* und *pipelining* verfügen. Der WCET-Analysierer AbsInt[7] von aiT nutzt das selbe Prinzip wie Bound-T berücksichtigt allerdings *caching* und *pipelining* für vorgegebene Prozessorarchitekturen.

Es ist nicht Ziel dieser Arbeit dieses Gebiet weiter zu vertiefen. Deshalb sei an dieser Stelle auf die Seminararbeit von Claudia Priesterjahn [15], im Rahmen dieses Seminars, verwiesen. Diese Arbeit setzt sich ins besondere mit den beiden oben genannten Verfahren auseinander.

4 Codegenerierung

Code-Generatoren sind zum Muss bei der modellbasierten Softwareentwicklung in der Wirtschaft geworden. Besonders durch den immer größer werdenden Zeitdruck den Unternehmen trotzen müssen, um weiterhin für den Markt existent zu bleiben. Das automatische Erzeugen von Code aus einem Modell hat diverse Vorteile, von dem der wichtigste Vorteil die Zeitersparnis ist. Wie einleitend in der Motivation erwähnt, entsteht eine bedeutende Zeitersparnis durch das Ersetzen der manuellen Programmierung mit einem automatischen Verfahren, welches nicht nur die Implementierung beschleunigt sondern Flüchtigkeitsfehler eines Programmierers vermeidet.

Aber was ist Codegenerierung? Um das Prinzip zu erläutern, kann man das Problem auf einer weitaus höheren Abstraktionsebene betrachten. Dabei wird von dem Begriff "Code" abstrahiert und es bleibt der Therm "Generierung". Eine Generierung ist ein Verfahren zur Erstellung eines "Gegenstandes". Die Beschaffenheit des "Gegenstandes" ist dabei zunächst irrelevant. Einer Generierung muss zudem immer eine Eingabe, als eine Art informative Spezifikation, vorliegen. Aus dieser informa-

[6] http://www.tidorum.fi/bound-t
[7] http://www.absint.com/ait

tiven Spezifikation soll eine Ausgabe generiert werden. Ein passenderer Begriff, im Bezug auf das Prinzip der Codegenerierung, wäre allerdings die Transformation, da es sich bei der Generierung immer um einen Prozess der Umwandlung handelt – zu Beginn steht die Idee / der Plan aus dem etwas geschaffen wird.

Das Prinzip der Transformation beschreibt die Codegenerierung auf einer weitaus abstrakteren Ebene, da hier von der Eingabe sowie der Ausgabe abstrahiert werden kann und der Prozess der Umwandlung einer Eingabe in eine Ausgabe den Fokus erlangt. Abbildung 3 zeigt schematisch das Prinzip der Codegenerierung aus der Perspektive der Transformation.

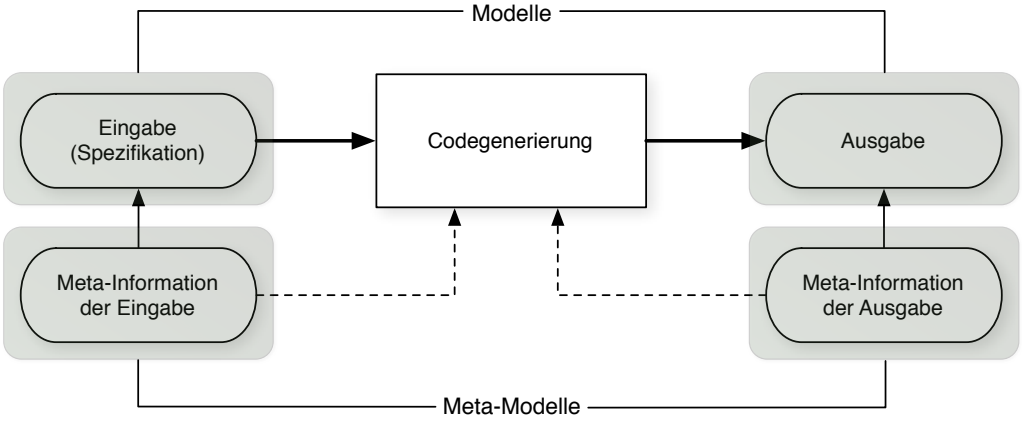

Abbildung 3: Schematisches Prinzip von Code-Generatoren

Um mit der Eingabe sowie der Ausgabe arbeiten zu können, werden diese allgemein als Modelle bezeichnet. Ein Modell ist hier allerdings nicht notwendiger Weise eine *domain specific language* (DSL), die von den MDD Ansätzen verwendet werden. Stattdessen kann ein Modell auch Code einer Hochsprache beschreiben. Für die Transformation werden allerdings noch weitere Informationen über die Modelle benötigt. Diese sind die so genannten Meta-Modelle. Meta-Modelle speichern Meta-Informationen, welche beschreiben wie ein Modell spezifiziert werden kann. Die Grammatik einer Sprache dient z.B. als Meta-Information zur Spezifikation einer bestimmten Sprache, um zu definieren wie Elemente der Sprache eingesetzt werden können.

Ein Verfahren mit dem zwei Modelle ineinander transformiert werden können, wird mit Hilfe von *abstract syntax trees* (AST) umgesetzt. Dabei wird das Eingabe-Modell zunächst mit einem *Parser* in einen AST konvertiert. Um einen AST für ein spezifisches Modell zu bilden, wird dass dazugehörige Meta-Modell benötigt. Mit einer Abbildung, welches eine Ansammlung von Transformations-Regeln einzelner Meta-Modell Informationen sind, kann dann aus dem AST der Eingabe ein AST für die Ausgabe erzeugt werden [17, 18]. Aus dem Ausgabe AST kann durch einen *Writer* Code erzeugt werden. Abbildung 4 zeigt das Verfahren für eine Java-Codegenerierung aus UML Klassendiagrammen.

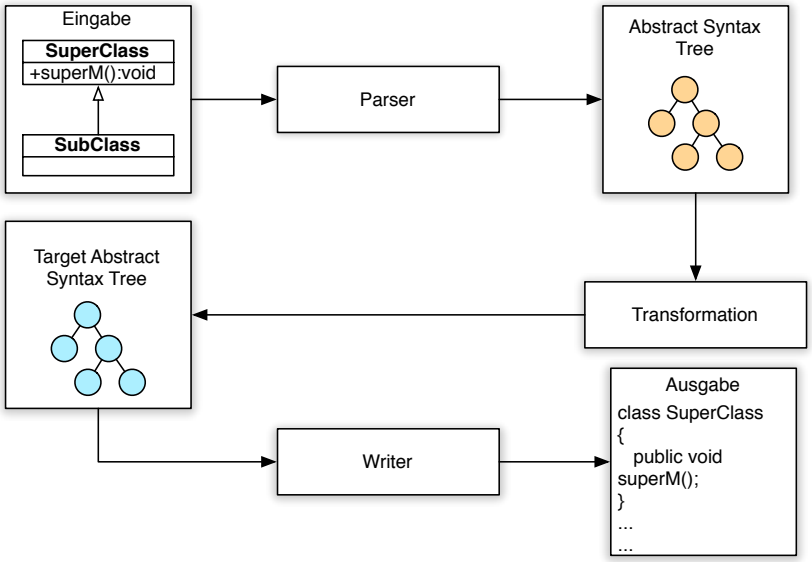

Abbildung 4: Modelltransformation als Grundlage für Codegenerierung

Der Prozess der von dem *Parser* sowie dem *Writer* durchgeführt wird, wird ebenfalls durch eine bijektive Abbildung zwischen dem Modell und Meta-Modell definiert, und ist bereits durch diese Abhängigkeit gegeben. Da bei Code-Generatoren die Ausgabe kompilierbarer Code einer Hochsprache ist und dafür i.d.R. keine expliziten Meta-Modelle vorliegen, wird bei dem *Writer* eher mit template- oder visitorbasierten Verfahren [6] gearbeitet. Diese übernehmen die Abbildung von AST in Code.

Code-Generatoren die allerdings nur eine direkte Model-zu-Code Transformation zulassen (Abbildung 4) sind, im Bezug auf ihre Modularität, inflexibel. Modularität bezieht sich hier auf die Möglichkeit zur einfachen Modifikation des Quellcodes, bezüglich bestimmten Plattformspezifika ohne großen Aufwand durch Modifikation des Code-Generators. Die OMG[8] hat einen Standard namens *Meta Object Facility* (MOF)[14] definiert, bei dem Modelle zwischen plattformunabhängig (PIM) sowie plattformabhängig (PSM) unterschieden werden. Ein PIM beschreibt ein Problem durch ein Modell, ohne dabei in irgendeiner Form plattformspezifisch zu sein. Das PSM ist ein Modell, dass das PIM um plattformspezifische Eigenschaften erweitert. Durch eine temporäre Model-zu-Model Transformation (PIM-zu-PSM) wird dann Code aus dem temporär transformierten PSM generiert. Mit Hilfe eines solchen Verfahrens können z.B. Optimierungen des Codes für bestimmte Plattformen einfacher umgesetzt werden [6].

[8] Object Management Group

4.1 Codegenerierung für zeitbehaftete Systeme

Aufgrund der steigenden Relevanz von Echtzeitsystemen und der damit verbundenen Relevanz von MDD in dieser Domäne steigt ebenfalls der Bedarf an Code-Generatoren für diese Domäne. Code-Generatoren für diese Domäne unterliegen zusätzlichen Restriktionen die auf Eigenschaften von Echtzeitsystemen zurückgeführt werden können, welche nun im folgenden erläutert werden. Diese Restriktionen beziehen sich dabei auf die Ausgabe, die diese Code-Generatoren erzeugen, welche allerdings bei nicht-zeitbehafteten Systemen, i.d.R., irrelevant sind. Von größerem Interesse sind hier besonders die *harten* Echtzeitsysteme, da diese restriktiveren Anforderungen unterliegen als *weiche* Echtzeitsysteme, wie bereits in Abschnitt 2 erläutert wurde.

Zunächst einmal wird von *harten* Echtzeitsystemen verlangt, dass diese nur dann eingesetzt werden wenn diese a-priori belegen können, dass das System zu keinem Zeitpunkt ein fehlerhaftes Verhalten aufweist. Fehlerhaftes Verhalten liegt dann vor wenn z.B. eine Berechnung die, von ihr geforderte, Deadline nicht einhält. Um dies zu verhindern kann eine *Schedulability*-Analyse durchgeführt werden welche, unter anderem, die WCET einzelner Prozesse benötigt. Dafür wird von dem Code erwartet, dass dieser Vorhersagbar ist. Der generierte Code darf aus diesem Grund keine Sprachkonstrukte verwenden, die einem Nicht-Determinismus folgen. Dies gewährleistet, dass der generierte Code messbar und damit als Eingabe für WCET-Analysen geeignet ist.

Da Echtzeitsysteme i.d.R. eingebettete Echtzeitsysteme sind, muss davon ausgegangen werden das die Ressourcen des Systems knapp sind. Dies führt zwangsläufig dazu, dass der generierte Code bezüglich der Ausführungszeit und der Größe optimiert sein sollte[9] [12]. Das Problem ist, dass man bei der Modellierung des Systems keinen Einfluss auf die Deadlines der einzelnen Prozesse hat, da diese direkt aus der Umwelt und der Situationen in der sich die Umwelt befindet, stammen. Wenn Prozesse ihre Deadlines nicht einhalten können gibt es also nur zwei Alternativen. Entweder man verwendet ein schnelleres System oder man optimiert den Code soweit, bis er die Deadline einhalten kann. Optimierter Code ist hier also von Vorteil um solche Änderungen im nach hinein zu vermeiden. Um Code-Optimierung berücksichtigen zu können werden allerdings detaillierte Informationen über den verwendeten *Compiler* sowie der Zielplattform benötigt. Dies macht die Codegenerierung damit im höchsten Maße zielplattform-spezifisch. Ein optimierter Code-Generator für eingebettete Echtzeitsysteme muss demnach für jegliche Konfiguration der Zielplattform konfiguriert werden können. Hier sind modulare Code-Generatoren von Vorteil die, mit möglichst kleinem Aufwand, die Anpassung an eine alternative Zielplattform erlauben.

Ein weiterer Aspekt ist, dass nicht immer davon ausgegangen werden kann, dass die Prozessoren der eingebetteten Echtzeitsysteme Fließkomma-Arithmetik un-

[9] Dies ist keine notwendige Bedingung für *harte* Echtzeitsysteme

terstützen. Für diesen Fall muss der Code einen eingeschränkten Sprachdialekt verwenden, der ohne Fließkomma-Variablen und Fließkomma-Operationen auskommt.

Wie bereits erwähnt unterliegt Software für *harte* Echtzeitsysteme schärferen Zeit- sowie Ressource-Bedingungen. Bei sicherheitskritischen Anwendungen liegt es zusätzlich im Interesse der Allgemeinheit wenn die Software Prüfungen unterliegt, die sicherstellen, dass zumindest der größte Teil der Fehler ausgeschlossen wird. Hierfür existieren, zumindest für den Automotive-Bereich, mehrere Verfahren um dies zu erreichen. Eine Möglichkeit ist, die Werkzeuge, die für den automatischen Prozess der Softwareerstellung zuständig sind, zu zertifizieren, was sogar durch die Norm RTCA DO-178B vorgeschrieben wird. Dies verbessert die Produkte in Hinsicht auf die Qualität, bezüglich der Fehlerrate. Die Zertifizierung von Code-Generatoren ist für die Praxis allerdings ein sehr kostspieliges und zeitintensives Verfahren, da der Code-Generator nur im Zusammenhang mit einer speziellen Konfiguration für eine spezifische Zielplattform zertifiziert wird. Eine kosten- und zeit-sparendere Variante ist die automatisch generierte Software allen Verifikationsstufen zu unterziehen, wie dies bei der manuellen Implementierung durchgeführt werden würde [4].

Als Beispiel Code-Generator für eingebettete Echtzeitsystem im Automotive-Bereich wird an dieser Stelle kurz der Code-Generator TargetLink von dSpace[10] erwähnt. TargetLink ist ein konfigurierbarer Code-Generator, der serienfähigen Code für den Einsatz in sicherheitskritischen eingebetteten Echtzeitsystemen im Automotive-Bereich erzeugt. TargetLink verwendet als Eingabe Modelle, die in der Erweiterung der Umgebung MatLab/Simulink[11], genannt Stateflow, definiert werden. Die Ausgabe von TargetLink ist ein spezifischer C-Dialekt namens MISRA C [13]. MISRA beschreibt 127 Restriktionen die der C-Code einhalten muss [4]. dSpace schreibt, dass dieser Code-Generator Code-Optimierungen für spezifische Zielplattformen durchführt, die, die Effizienz von handgeschriebenen Code übersteigen kann (siehe Abbildung 5).

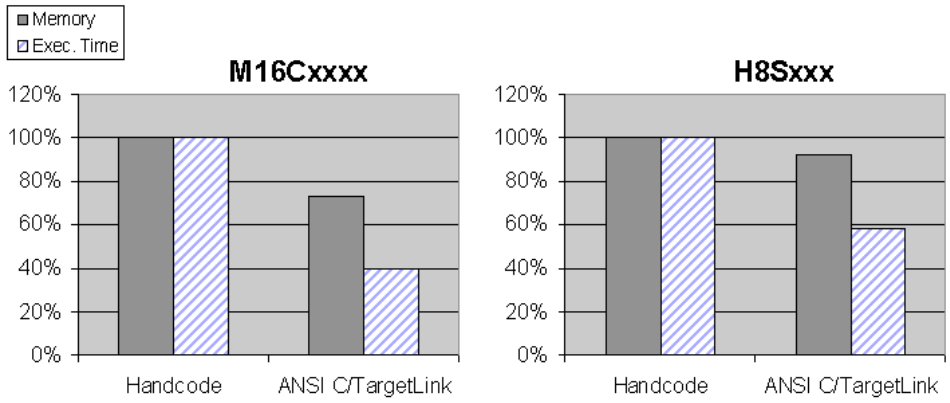

Abbildung 5: Benchmark zum Vergleich der Effizienz von TargetLink- und handgeschriebenem-Code [4]

[10] http://www.dspace.org
[11] http://www.mathworks.com

4.1.1 Codegenerierung für kontinuierliche Systeme

Ein kontinuierliches Echtzeitsystem beschreibt auf feinster granularer Ebene ein System welches, durch eine Eingabe und einer Berechnung (Transformation) auf der Eingabe, eine entsprechende Ausgabe erzeugt. Die Eingabe ist dabei ein kontinuierlicher Wert, der entweder über Sensoren aus einer physikalischen Umgebung stammt oder die Ausgabe einer anderen Berechnung ist. Die Berechnung beschreibt das kontinuierliche Verhalten eines solchen Systems. Eine schematische Darstellung eines einfachen Systems ist in Abbildung 6 illustriert.

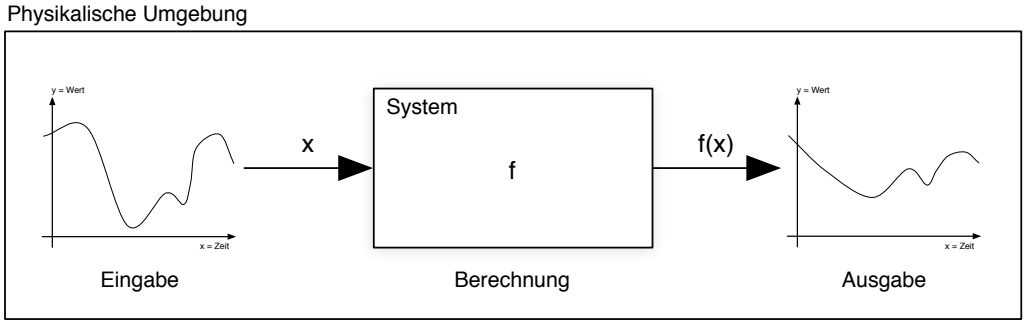

Abbildung 6: Kontinuierliches reaktives Echtzeitsystem

Das System, dass eine Berechnung auf der Eingabe x durchführt, wird als Differentialgleichung f definiert. Die Ausgabe $f(x)$ ist das Ergebnis der Berechnung f auf der Eingabe x. Zusätzlich sollte ein reaktives Echtzeitsystem dafür sorgen, dass bei Ausfall der Eingabe, zumindest versucht wird, eine korrekte Ausgabe zu berechnen. Dies kann z.B. durch die Verwendung heuristischer Daten der Eingabe aus der Vergangenheit, zumindest für eine gewissen Überbrückungszeit, erzielt werden. Reaktive Echtzeitsysteme terminieren nicht sondern führen ihre Berechnungen kontinuierlich auf der Eingabe x aus.

Da Computersysteme i.d.R. nur quasi kontinuierlich operieren, kann ein kontinuierliches Echtzeitsystem nie auf einem diskreten Computersystem realisiert, sondern nur beliebig genau approximiert werden. Das Problem ist zum Einen die Zeit-Diskretisierung die durch die Taktung des Systems entsteht, und zum Anderen die Wert-Diskretisierung die durch den endlichen diskreten Wertbereich entsteht. Abbildung 7 zeigt das Problem anhand einer Grafik.

Dies sind substantielle Probleme auf die ein Code-Generator für diese Domäne eingehen muss. Für das Problem der Zeit- und Wert-Diskretisierung kann der Code so generiert werden, dass zum Einen die Taktung erhöht wird, was natürlich durch die maximale Taktung des Systems beschränkt ist, und zum Anderen in dem mehr Speicher für den Wertbereich von Variablen verwendet wird um eine präzisere Darstellung der Werte zu erlangen. Beides sind Methoden um eine genauerer Approximation des beschriebenen kontinuierlichen Verhaltens zu erlagen.

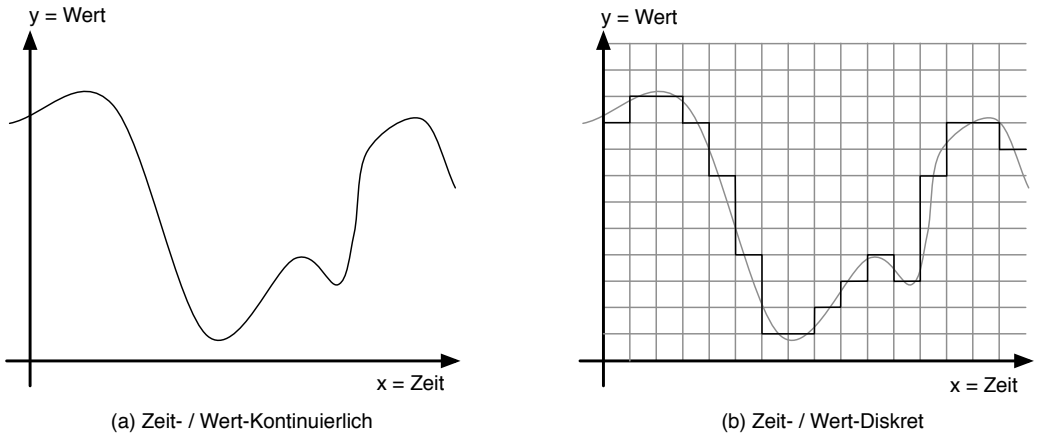

Abbildung 7: Zeit- / Wert-Kontinuierlich vs. Zeit- / Wert-Diskret

Ein anderer wichtiger Aspekt ist die effiziente Darstellung von differenzierbaren Funktionen f in Code. Für die Darstellung kontinuierlicher Differentialgleichungen in Code werden i.d.R. zunächst approximierte Verfahren wie z.B. die *Euler*-, *Heun*- oder *Runge-Kutta*-Näherungsverfahren verwendet um eine approximierte Darstellung des kontinuierlichen Verhaltens zu erlangen. Danach muss aus dieser approximierten Darstellung der Code möglichst effizient generiert werden. Die angegebenen Näherungsverfahren unterscheiden sich in ihrem Fehler, den sie bereits in der ersten Ordnung aufweisen. So besitzt das *Euler*-Näherungsverfahren einen Fehler von $O(\Delta^2)$, *Heun*- $O(\Delta^4)$ und das *Runge-Kutta*-Näherungsverfahren sogar nur $O(\Delta^5)$ wobei Δ den Abstand zwischen zwei Messpunkten definiert, der sich durch die Taktung der Berechnung ergibt.

Trotz all dieser Lösungen wird das ursprüngliche Problem auf solchen Computersystemen immer nur approximiert. Der erzeugte Code stimmt demnach nie zu 100% mit dem Modell überein was bei der Implementierung der Problems berücksichtig werden muss, um nicht Fehler durch eine solche Approximation zu forcieren. Das bedeutet nicht, dass solche Systeme nicht implementiert werden können, sondern dass es sich dabei immer um hinreichende Approximationen handelt.

4.1.2 Codegenerierung für diskrete Systeme

Diskrete Echtzeitsysteme sind auf der Zeit- sowie Wert-Achse diskret allerdings auf einer weitaus abstrakteren Ebene. So ist mit diskreter Zeit bei den diskreten Echtzeitsystemen nicht die Rasterung einer kontinuierlichen Zeit zu verstehen, sondern das Unterteilen in diskrete Zustände, in denen sich das System befinden kann. Auch das Verhalten des Systems kann nicht als einfache Diskretisierung kontinuierlichen Verhaltens betrachtet werden, sondern muss auch hier auf einer abstrakteren Ebene betrachtet werden. Diskretes Verhalten kann als Funktion beschrieben werden, welche komplexe Modifikationen an dem System (Änderungen von Variableninhalten),

bis hin zu diskreten Zustandsänderungen, sein können. Computerprogramme, wie sie aus dem Home- / Office-Bereich bekannt sind, sind i.d.R. diskrete Systeme.

Die Modellierung solcher diskreten Echtzeitsysteme kann mit Hilfe von zustandsbasierten Modellen erfolgen wie z.B. Statecharts. Die Implementierung eines diskreten Echtzeitsystems kann mit Hilfe zwei unterschiedlicher Ansätze realisiert werden. Der einfachere Ansatz ist die Realisierung eines synchronen Echtzeitsystems. Bezüglich dieser Systeme muss die Codegenerierung ein soliden Rahmen generieren, der die Abbildung der Zustandsänderungen eins zu eins in Code transferiert. Zustandsübergänge werden in diskreten Modellen i.d.R. mit einer Schaltzeit von Null definiert. Der Code-Generator muss an dieser Stelle Code generieren, der dieser Semantik entspricht. Dies ist allerdings in der Praxis nicht möglich. Eine diskreter Zustandsübergang kann im Code z. B. durch ein einfaches if-then-else oder switch-case Konstrukt implementiert werden. Diesbezüglich wurde in einigen Modellen, wie z.B. Real-Time Statecharts, die Möglichkeit integriert, Benutzerdefinierte Schaltzeiten für Übergänge zu definieren. Dadurch kann das Modell in Übereinstimmung mit dem generierten Code gebracht werden [7].

Zustandsübergänge können allerdings nicht ausschließlich durch Aktionen initiiert werden, sondern auch durch das Vergehen von Zeit oder in Kombination mit einer vorher eingetretenen Aktion. Ist das Schalten eines Übergangs abhängig von dem Vergehen von Zeit, so führt dies zu Grundlegenden Problemen, die auf die Diskretisierung von Zeit zurückzuführen sind. Wird z.B. ein Übergang dadurch definiert, dass eine Bedingung $x = 11$ erfüllt sein muss, wobei x eine Uhr ist, so kann es passieren, dass das generierte System eventuell nie bzw. nicht vorhersagbar Schalten wird, da die diskreten Zeitschritte des Systems nie diesen Wert erreichen könnten. Für diese Art von Problem existieren Verfahren die diese Fehler kompensieren. Ein Beispiel für ein solches Verfahren wird von Arnand *et al.* in [1] erläutert. Das Verfahren bezieht sich in dem Papier zwar auf hybride Systeme kann aber ebenfalls für diskrete Systeme in Betracht gezogen werden. In dem Papier wird das hybride Modell mit Hilfe einer Spezifikation des Aktualisierungszyklus des Systems analysiert um ein zwischen Modell zu erzeugen bei dem mögliche verpasste Übergänge berücksichtig werden.

Synchrone Systeme sind durch ein geschlossenes Programm definiert, d.h. Zustandsübergänge werden ausschließlich durch lokale Berechnungen initiiert. Asynchrone Systeme bergen dagegen mehr Gefahrenpotential, wie unerwartete Seiteneffekte, die durch Parallelisierung entstehen können. Die Interaktion bei asynchronen Systemen ist weitaus komplexer als bei synchronen, da diese i.d.R. verteilte Echtzeitsysteme sind die über Nachrichten interagieren. Komplexere Echtzeitsysteme sind in der Praxis ebenfalls verteilte Echtzeitsysteme, wie z.B. die Software in Automobilen.

4.1.3 Codegenerierung für hybride Systeme

Hybride Echtzeitsysteme haben den Vorteil, dass diese die Vorteile der beiden zuvor erwähnten Echtzeitsysteme vereinen. Echtzeitsysteme im Bereich der Regelungstech-

nik machen einen Großteil der eingebetteten Echtzeitsysteme aus. Diese müssen, aufgrund der regulierenden Aufgabe die diese Echtzeitsysteme zu erfüllen haben, zumindest kontinuierliche Echtzeitsysteme sein da Informationen der Umwelt über Sensoren erfasst werden und dann quasi kontinuierlich verarbeitet werden müssen. Damit scheiden diskrete Echtzeitsysteme für das Einsatzgebiet der Regelungstechnik aus.

Dagegen können Hybride Echtzeitsysteme für regelungstechnische Aufgaben eingesetzt werden da diese, wie auch kontinuierliche Echtzeitsysteme, kontinuierliches Verhalten besitzen. Der große Vorteil liegt in den zusätzlichen diskreten Zuständen die das System zusätzlich besitzt [3]. Zu definierten Zeitpunkten kann ein solches System seinen Zustand verändern, in dem es diskrete Zustandsübergänge durchführt. Da jeder diskrete Zustand mit kontinuierlichem Verhalten gekoppelt ist, beschreibt, bei parallelen Systemen, die Menge aller aktiven Zustände oder, bei sequentiellen Systemen, der aktive Zustand das Gesamtverhalten eines Systems. Anders als bei reinen kontinuierlichen Systemen kann ein hybrides System sein Gesamtverhalten zur Laufzeit modifizieren.

Beschrieben werden diese Systeme, wie auch die diskreten Echtzeitsysteme durch zustandsbasierte Modelle dessen Zustände allerdings durch Blockdiagramme beschrieben werden, welche kontinuierliches Verhalten über verschaltete Differentialgleichungen beschreiben. Modelliert werden können solche Systeme z.B. durch Hybrid Automata [9] in dem jeder Zustand durch eine differenzierbare Funktion beschrieben wird.

Aber nicht nur die Vorteile der kontinuierlichen sowie diskreten Echtzeitsysteme werden hier vereint sondern auch die Probleme, mit denen die Code-Generatoren sich auseinandersetzen müssen. So muss bei dem kontinuierlichen Berechnungen darauf geachtet werden, dass Differentialgleichungen mit einem Trade-Off zwischen Effizienz und Genauigkeit umgesetzt werden. Bei dem diskreten Teil muss bei zeitgesteuerten Übergängen darauf geachtet werden, dass der generierte Code in der Ausführung keine Übergänge verpasst.

Code-Generatoren für diese Art von Echtzeitsystemen müssen die Eigenschaften der Code-Generatoren von kontinuierlichen Echtzeitsystemen, bezüglich der Generierung der Differentialgleichungen, und von diskreten Echtzeitsystemen, bezüglich der Zustandsänderungen, vereinen.

5 Zusammenfassung

Modellbasierte Softwareentwicklung hat sich als wichtiges Werkzeug, bei der Softwareentwicklung von softwareintensiven Systemen in der Wirtschaft, etabliert. Ein ebenso relevanter Aspekt von modellbasierten Entwicklungswerkzeugen sind die Code-Generatoren, welche es dem Softwareentwickler ermöglichen Code direkt aus dem Modell zu erzeugen ohne dafür von Programmierern abhängig zu sein. Code-Generatoren

führen damit die Implementierung, innerhalb der eigentlich kostenintensiven Phase, automatisch durch.

Code-Generatoren finden ausserdem immer weiteren Anklang in der MDD Domäne für eingebettete Echtzeitsysteme. Code-Generatoren für diese Domäne unterliegen zusätzlichen Restriktionen, bezüglich des generierten Codes, was das Entwickeln von Code-Generatoren in dieser Domäne erschwert allerdings nicht unmöglich macht. Stateflow mit dem Code-Generator TargetLink von dSpace, welche auf MatLab/Simulink aufbauen, haben gezeigt, dass qualitativ hochwertiger Code für Anwendungen in dieser Domäne durchaus erzeugt werden kann und wird.

Da diese Domäne immer mehr Relevanz in der Wirtschaft erlangt, durch steigende Integration von Regelungstechnik in allgegenwärtigen Technologien, wird in Zukunft der Bedarf an kompetenten Code-Generatoren für diese Domäne steigen. Die Kompetenz eines Code-Generators hängt dabei von der Variablen Unterstützung der Zielplattformen und der Qualität des Codes ab, wobei die Qualität anhand verschiedener Attribute, wie z.B. der Fehlerrate, gemessen werden kann.

Literatur

[1] M. Anand, J. Kim, and I. Lee. Code generation from hybrid systems models for distributed embedded systems. In *ISORC*, pages 166–173. IEEE Computer Society, 2005.

[2] J. Apsel. *Ausführbarkeitstests und Scheduling-Algorithmen in dynamischen Mehrprozessorsystemen.* PhD thesis, Technische Universität Clausthal, 1998.

[3] P. I. Barton and C. K. Lee. Modeling, simulation, sensitivity analysis, and optimization of hybrid systems. *ACM Trans. Model. Comput. Simul.*, 12(4):256–289, 2002.

[4] M. Beine and M. Jungmann. Einsatz von automatischer Code-Generierung für die Entwicklung von sicherheitskritischer Software.

[5] G. C. Buttazzo. *Hard Real-Time Computing System: Predictable Scheduling.* Academic Publishers, 2002.

[6] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *In online proceedings of the 2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA. Anaheim*, October 2003.

[7] E. Erpenbach. *Compilation, Worst-Case Execution Times, and Scheduling Analysis of Statecharts Models.* Embedded Systems. C-LAB Pulications, 2000.

[8] D. Gillies. What exactly is meant by real-time? Real-Time Magazine, FAQ: www.realtime-info.be.

[9] T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS '96)*, pages 278–292, New Brunswick, New Jersey, 1996.

[10] P. Hsiung. Formal Synthesis and Code Generation of Embedded Real-Time Software. pages 208–213.

[11] R. Kirner and P. P. Puschner. Classification of WCET analysis techniques. In *ISORC*, pages 190–199. IEEE Computer Society, 2005.

[12] R. Leupers. Code generation for embedded processors. In *ISSS '00: Proceedings of the 13th international symposium on System synthesis*, pages 173–178, Washington, DC, USA, 2000. IEEE Computer Society.

[13] MISRA. Guidelines for the Use of the C Language in Vehicle Based Systems, April 1998.

[14] Object Management Group. *Meta Object Facility 1.4, OMG Document: formal/02-04-03*, 2003.

[15] C. Priesterjahn. Worst-Case-Execution Analyzer. Universität Paderborn. Sommersemester 2006. Seminar: Software Engineering für softwareintensive Systeme, August 2006.

[16] G. Smith. Specifying mode requirements of embedded systems. In *ACSC '02: Proceedings of the twenty-fifth Australasian conference on Computer science*, pages 251–257, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[17] L. Tratt. Th MT model transformation language. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1296–1303, New York, NY, USA, 2006. ACM Press.

[18] H. Vangheluwe and J. de Lara. Foundations of multi-paradigm modeling and simulation: computer automated multi-paradigm modelling: meta-modelling and graph transformation. In *WSC '03: Proceedings of the 35th conference on Winter simulation*, pages 595–603. Winter Simulation Conference, 2003.

[19] I. Wenzel, B. Rieder, R. Kirner, and P. Puschner. Automatic timing model generation by cfg partitioning and model checking. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 606–611, Washington, DC, USA, 2005. IEEE Computer Society.