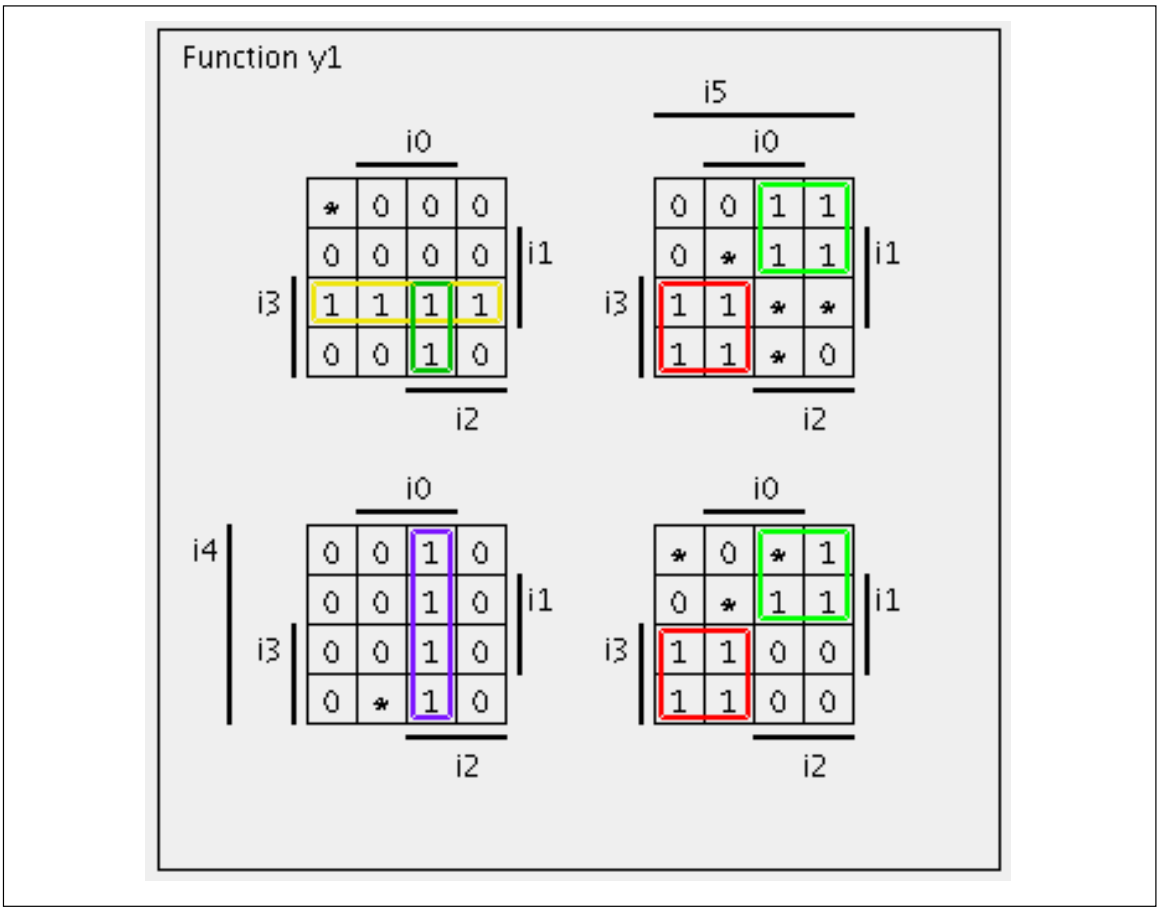
Projektbericht 09/2004

Das interaktive Skript

Automatische Überprüfung und Hilfestellung zu Vorlesungs-begleitenden Übungen

Klaus von der Heide, Norman Hendrich
Universität Hamburg
Fachbereich Informatik



E-LEARNING CONSORTIUM Hamburg

Inhaltsverzeichnis

Kontakt:
Prof. Dr. Klaus von der Heide
Universität Hamburg
Fachbereich Informatik
Vogt-Kölln-Str. 30
D 22 527 Hamburg

http://tams-www.informatik.uni-hamburg.de/forschung/interaktives-skript/

1 Einführung

Der vorliegende, dritte Projektbericht des Projekts „Automatische Überprüfung und Hilfestellung zu Vorlesungs-begleitenden Übungen“ behandelt Erweiterungen unserer Software-Infrastruktur und erläutert weitere jetzt konkret umgesetzte Verfahren zur Überprüfung von Aufgaben.

Einführung...

Als Einführung fasst der folgende Abschnitt 1.1 noch einmal kurz die wesentlichen Aspekte eines *interaktiven Lehrbuchs* zusammen, während Abschnitt 1.2 die Einbettung von Übungsaufgaben motiviert. In Abschnitt 1.3 wird noch einmal die in der ersten Projektphase erarbeitete Klassifikation der Übungsaufgaben wiederholt.

Die Gliederung des vorliegenden Projektberichts wird in Abschnitt 1.4 erläutert. Für eine ausführliche Diskussion der mit den interaktiven Lehrbüchern verfolgten Konzepte sei auf den ersten Projektbericht [11] verwiesen.

und Vertiefung

Als Ergänzung der Beschreibungen wird in den folgenden Kapiteln bei Bedarf auf kurze Programmbeispiele und Skripte zurückgegriffen, um die zugrundeliegenden Konzepte zu erläutern. Dabei werden auch einige Schnittstellen der verschiedenen Funktionen vorgestellt, so dass der Bericht gleichzeitig als Softwaredokumentation genutzt werden kann. Die entsprechenden Abschnitte dienen zur Vertiefung und können beim Lesen ohne weiteres übersprungen werden.

1.1 Rahmen des Projekts: Das interaktive Lehrbuch

Konzept

Ein *interaktives Lehrbuch* bzw. *interaktives Skript* vereinigt die textuelle Beschreibung der zu lernenden Zusammenhänge und Sachverhalte mit interaktiven elektronischen Werkzeugen zur Darstellung und Anwendung dieser Sachverhalte — und zwar der Anwendung nicht nur auf die im Lehrbuch selbst integrierten Beispiele, sondern vielmehr auf beliebige, vom Lernenden jederzeit selbst veränderbare oder hinzugefügte Anwendungsfälle. Dadurch wird das Lehrbuch erweiterbar und kann auch an ursprünglich gar nicht vorgesehene oder vorhersehbare zukünftige Entwicklungen angepasst werden. Es bietet damit ideale Voraussetzungen zur Unterstützung des *life-long learning* und zum produktiven dauerhaften Einsatz während des Berufslebens.

Das dem Projekt zugrunde liegende Konzept des interaktiven Lehrbuchs hat folgende Zielsetzungen:

- *Problemlösung*

  Die Spanne zwischen klassischem Lehrbuch und Anwendung wird zunehmend größer, weil die Komplexität der behandelten Themen sich nur noch mit Rechner-gestützten Systemen beherrschen lässt. Durch die harmonische Integration von klassischem Lehrtext in ein zugrunde liegendes, universelles Softwaresystem zur Problemlösung kann das interaktive Lehrbuch sehr anwendungsspezifisch werden, ohne dabei jedoch auf ein Anwendungsgebiet festgelegt zu sein.

- *Nachhaltigkeit*

  Der Rückblick auf die letzten Jahre der Softwareentwicklung zeigt, dass für nachhaltige Entwicklungen nur einfache und standardisierte Datenformate

verwendet werden sollten. Beim Einsatz proprietärer Formate muss jederzeit damit gerechnet werden, nach einem Versionswechsel auf ältere Datensätze nur noch eingeschränkt oder eventuell überhaupt nicht mehr zugreifen zu können. Inhalte für das interaktive Lehrbuch basieren daher im Wesentlichen auf annotierten Texten im einfachen ASCII-Format.

Anpassbarkeit

- Ein klassisches Lehrbuch spiegelt immer nur die Sicht (und Absicht) des jeweiligen Autors wider. Um sich ein objektiveres Bild zu verschaffen, greifen viele Studierende und Lehrende deshalb parallel auf mehrere Lehrbücher zurück. Wegen des hohen Erstellungsaufwands im Falle von E-Learning Content stehen geeignete alternative Materialien bisher aber nur selten zur Verfügung. Daraus ergibt sich die Forderung, dass der Inhalt des interaktiven Lehrbuchs von den Lehrenden individuell nach ihren eigenen Vorstellungen ausgerichtet und geändert werden kann — und zwar mit Hilfe eines langfristig und auf allen Plattformen verfügbaren Werkzeugs. Dies betrifft sowohl die Auswahl als auch die Inhalte der Texte, Abbildungen, Animationen, Audioausgaben, Simulationen, usw.

Exploration

- Die Integration von interaktiven Elementen in das interaktive Lehrbuch hilft dabei, Interpretationsschwierigkeiten oder Verständnislücken durch aktive Exploration des Lehrstoffs zu überwinden. Viele Inhalte und insbesondere Graphiken im interaktiven Lehrbuch werden deshalb erst zur Laufzeit mit vom Benutzer individuell einstellbaren Parametern erzeugt und angezeigt. Ein Studierender kann auf diese Weise grundsätzlich nachvollziehen, wie es zu den Graphiken und Ergebnissen kommt.

1.2 Kurzbeschreibung des Projekts

Integrierte Übungen

Es liegt nahe, auch *Übungsaufgaben* in das oben beschriebene interaktive Lehrbuch zu integrieren. Während eine derartige Integration bei klassischer Lernsoftware im Sinne des *Computer Based Training* [26] sehr aufwendig sein kann, stellt das interaktive Lehrbuch mit seiner Softwareplattform bereits alle Werkzeuge zur Verfügung, um die Lernenden bei der Bearbeitung der Aufgaben zu unterstützen und zu motivieren.

Überprüfung und Hilfestellung

Die Erforschung und Erprobung dieser Techniken ist der Inhalt des vorliegenden Projekts. Ziel ist eine Softwarebibliothek, die für viele Typen von Übungsaufgaben eine automatische Überprüfung der Lösungen erlaubt und bei Fehlern geeignete Hilfestellung anbietet. Die Studierenden bekommen dadurch sofort eine Rückmeldung über ihren Lernfortschritt bzw. Hinweise auf noch verbliebene Fehler. Zusammen mit der nahtlosen Integration der Aufgaben in das Skript wird die Hemmschwelle zur Bearbeitung der Übungsaufgaben gesenkt, was die Studierenden zu einer intensiveren Beschäftigung mit dem Lehrstoff einlädt.

Methodenkompetenz

Das primäre Ziel der Übungen ist dabei nicht die Vermittlung von Faktenwissen, sondern die Verbesserung der Fertigkeiten bei der Auswahl und dem Einsatz von Methoden. Die im Projekt erzielten Ergebnisse werden sich deshalb auch auf viele andere Fachgebiete mit ähnlicher Methodik übertragen lassen, etwa die angewandte Mathematik, Experimentalphysik oder die Ingenieurwissenschaften.

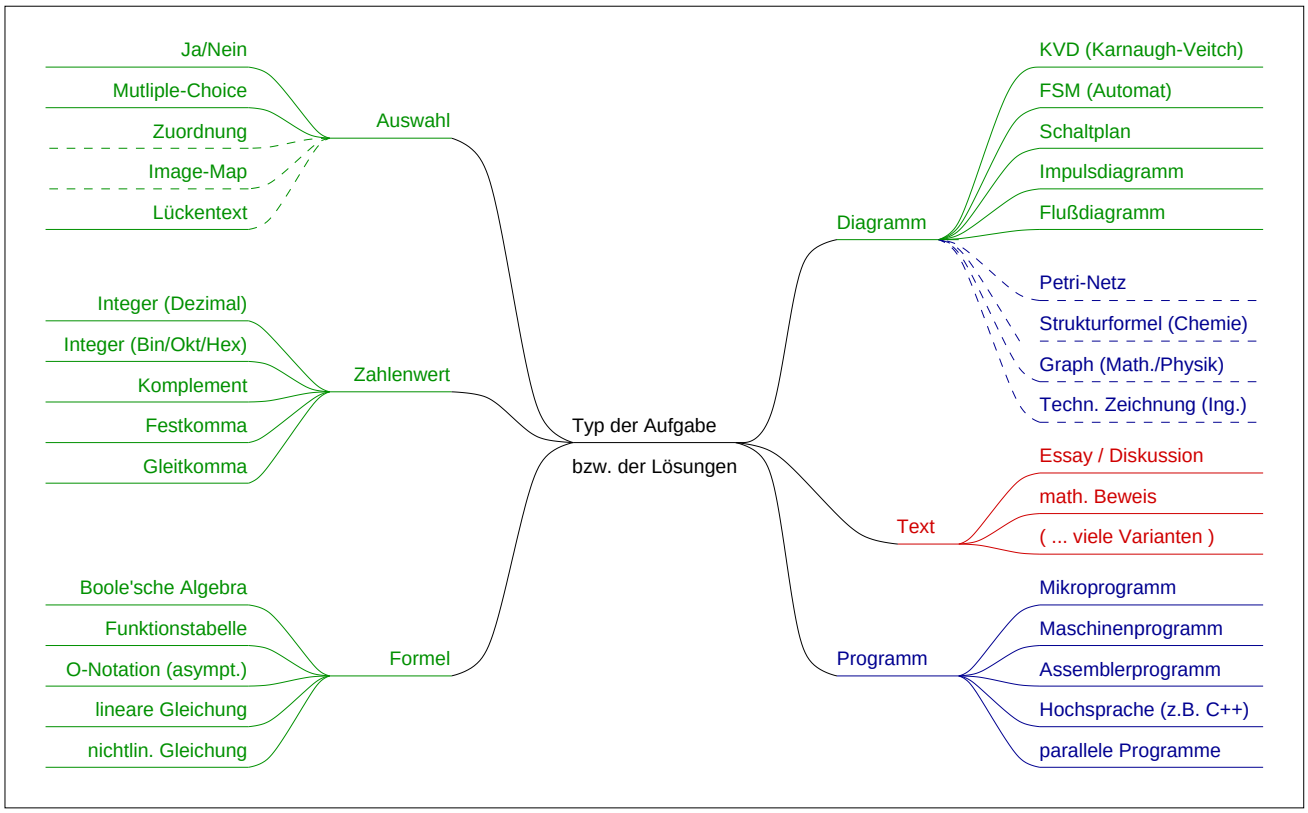


Abbildung 1: *Klassifikation der Übungsaufgaben zur technischen Informatik. Die sechs Hauptklassen dürften auch auf andere mathematisch naturwissenschaftliche Fächer übertragbar sein. Für fast alle Kategorien bis auf freie Texte ist eine automatische Überprüfung möglich.*

1.3 Klassifikation der Übungsaufgaben

An dieser Stelle ist es notwendig, die in der ersten Projektphase erarbeitete Klassifikation der Übungsaufgaben zur technischen Informatik noch einmal zu wiederholen, da die späteren Kapitel häufig auf diese Klassifikation zurückgreifen. Die Auswertung mehrerer Vorlesungsskripte und klassischer Lehrbücher ergab die in Abbildung 1 dargestellte Einteilung in sechs große Klassen von Aufgabentypen. Diese Klassen von Aufgaben dürften sich auch auf die meisten anderen technisch-naturwissenschaftlichen Fächer übertragen lassen, allerdings eventuell mit anderer Gewichtung und Häufigkeit der einzelnen Aufgabentypen.

Wie bereits im ersten Projektbericht erläutert wurde, ist es beim derzeitigen Stand der Technik zur Texterkennung und Sprachverarbeitung nicht einmal ansatzweise möglich, von den Studierenden eingesandte freie Texte sinnvoll auszuwerten [11]. Im Rahmen des Projekts wird diese Kategorie deshalb von vornherein ausgeklammert; derartige Aufgaben müssen wie bisher von den Übungsgruppenleitern von Hand korrigiert werden. Statt dessen ist geplant, die Textaufgaben zumindest soweit möglich durch gleichwertige Fragestellungen zu ersetzen, die der automatischen Überprüfung besser zugänglich sind.

1.4 Gliederung dieses Berichts

Kapitel 2 Das folgende Kapitel 2 erläutert noch einmal die für das Projekt gewählte Software-Architektur. Als eigentliche interaktive Softwareumgebung kommen dabei sowohl Matlab als auch die Kombination von Jython und Java zum Einsatz.

Gegenüber der in den vorherigen Reports vorgestellten Architektur sind zwei Neuerungen zu beachten. Zum einen eröffnet die seit kurzem verfügbare Version R14 von Matlab die Möglichkeit, Teile der interaktiven Skripte zu compilieren und damit unabhängig von Matlab-Lizenzen einsetzen zu können. Zum anderen konnte das im letzten Report skizzierte Konzept zur Umsetzung der Skripte als HTML-Seiten mit kleinen eingebetteten Java-Applets realisiert werden. Damit können alle gängigen Web-Browser zur Darstellung genutzt werden und es steht eine Alternative zum bisher verwendeten *mscriptview*-Browser zur Verfügung.

Kapitel 3 Kapitel 3 skizziert die Auswertung der Klasse der *Formelaufgaben*. Hier werden zunächst die verschiedenen Varianten skizziert, die im Rahmen der Vorlesungen und Übungsaufgaben zur Technischen Informatik vorkommen. Wegen der zentralen Bedeutung für den Entwurf digitaler Systeme bildet dabei die Überprüfung von Boole'schen Ausdrücken und Funktionen einen besonderen Schwerpunkt.

Kapitel 4 Kapitel 4 beschreibt unseren Ansatz zur Überprüfung von *Maschinen- und Assemblerprogrammen* für unseren Demonstrations-Rechner PRIMA. Da es auf diesen Abstraktionsebenen mangels Typinformation praktisch unmöglich ist, eine brauchbare semantische Analyse der Programme durchzuführen, setzen wir wieder auf die Kombination von Simulation mit Plausibilitätstests.

Der Bericht schließt mit einer kurzen Zusammenfassung und dem Literaturverzeichnis.

2 Infrastruktur und Plattformen

Die Grundlage für das Konzept der interaktiven Skripte ist eine Softwareplattform, die ein interaktives Ausführen von im Skript selbst eingebettetem Programmcode gestattet. Die von uns gewählte Softwarearchitektur basiert auf einer Kombination der beiden Plattformen *Matlab* und *Java*. Da gegenüber dem letzten Projektstand zwei wichtige Erweiterungen implementiert bzw. erprobt werden konnten, sollen diese in den nächsten Abschnitten vorgestellt werden.

Im Rahmen des Projekts wurden mehrere Varianten untersucht, von denen sich die folgenden vier Kombinationen als besonders geeignet herausgestellt haben:

- Browser *mscriptview*. Zum Lesen der Skripte ist eine Matlab-Installation mit Lizenz erforderlich, aber die Content-Erstellung ist besonders einfach. Dieser Ansatz wird in Abschnitt 2.1 noch einmal zusammengefasst.
- HTML-Browser mit Applet-Schnittstelle zu Matlab. Texte und passive Elemente wie Graphiken sind uneingeschränkt nutzbar, aber für das Ausführen der aktiven Elemente ist eine Matlab-Installation mit Lizenz erforderlich. Diese Architektur wird in Abschnitt 2.3 vorgestellt.
- HTML-Browser mit Applet-Schnittstelle zu vorcompilierten externen Applikationen. In Abschnitt 2.4 wird gezeigt, wie sich auch unsere Matlab-Funktionen auf diese Weise nutzen lassen.
- HTML-Browser mit Applet-Schnittstelle zu Jython als Skriptsprache an Stelle von Matlab, siehe Abschnitt 2.5. Frei verfügbar, aber Skripte bisher nur teilweise umgesetzt.

Diese Aufteilung ist in Abbildung 2 noch einmal veranschaulicht. Natürlich können die verschiedenen Ansätze auch kombiniert werden; so setzen die Skripte zur Vorlesung T1 weiterhin auf aktive Matlab-Elemente, während die meisten Algorithmen zur Überprüfung von Übungsaufgaben in Java/Jython implementiert wurden.

Die im letzten Projektbericht erläuterte Variante mit einem proprietären, um zusätzliche Syntaxelemente erweiterten XML-Browser wurde übrigens nicht weiterverfolgt, da sich die Implementierung als zu aufwendig erwies. Das Problem der aufwendigen Content-Erstellung in HTML konnte durch den in Abschnitt 2.3 beschriebenen Konverter gelöst werden.

2.1 Plattform Matlab mit Browser mscriptview

Matlab

Als Plattform für die bisher erstellten interaktiven Skripte dient Matlab, das international führende Softwaresystem für numerische Mathematik mit Schwerpunkt auf Anwendungen in technisch-orientierten Fachgebieten [21]. Mit dem Begriff Matlab werden dabei sowohl das Gesamtsystem als auch die zugrunde liegende Programmiersprache bezeichnet, die sich durch eine besonders einfache Formulierung von Matrixoperationen auszeichnet. Als Ergänzung zum Grundsystem stehen diverse Erweiterungen bereit, die applikationsspezifische Funktionen in sogenannten Toolboxes bzw. Blocksets zusammenfassen. Die Software steht nicht nur für Windows,

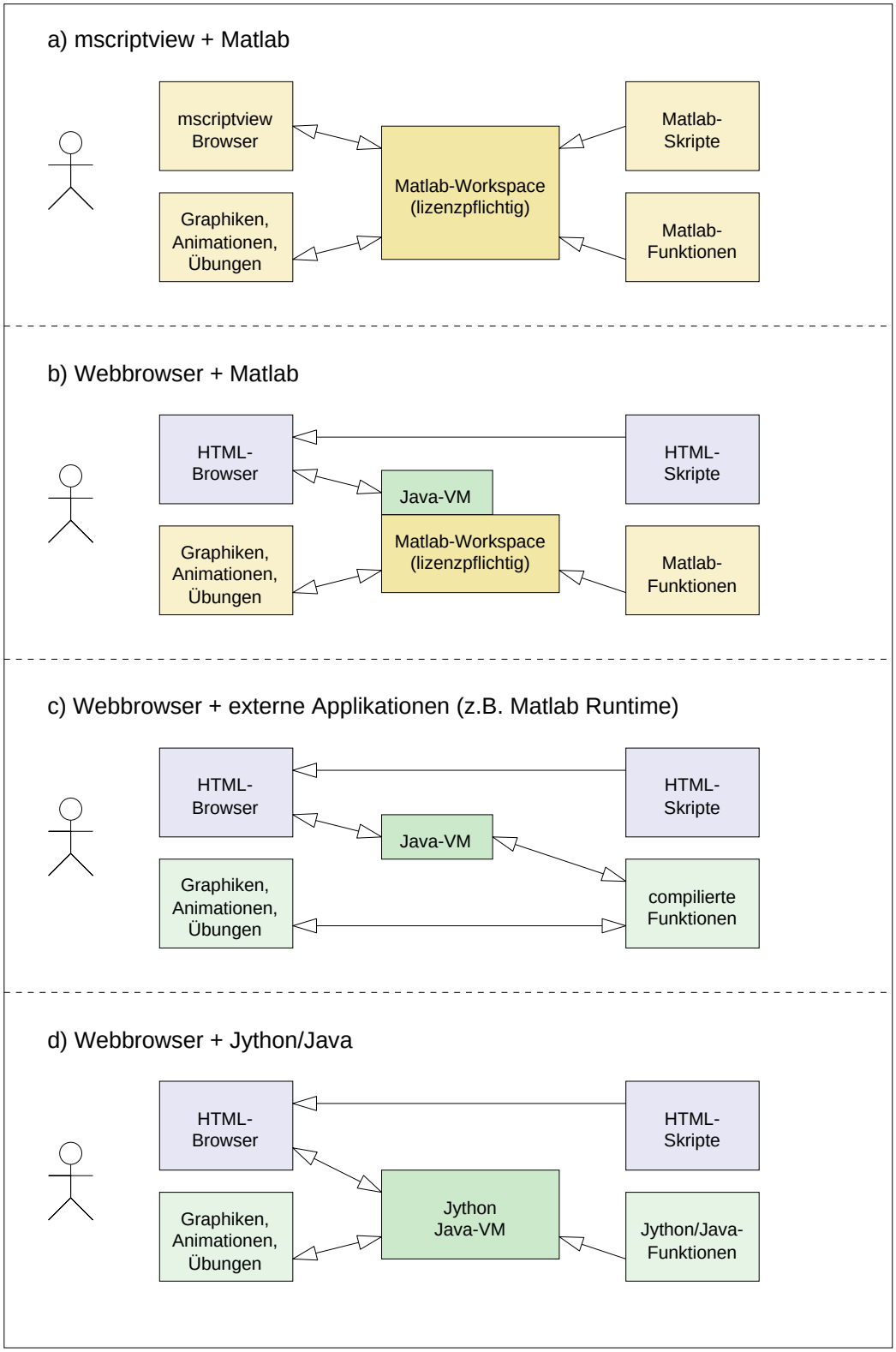


Abbildung 2: *Die Abbildung zeigt vier mögliche Varianten zur Realisierung interaktiver Skripte mit den Plattformen Matlab sowie Java/Jython: a) Verwendung von Matlab und Darstellung mit mscriptview-Browser, b) Einbindung von Matlab-Funktionen in HTML via Applets, c) Aufruf vorcompilierter (Matlab-) Applikationen aus HTML via Applets, d) Einbindung von Jython-Applets. Natürlich können die verschiedenen Varianten auch kombiniert werden.*

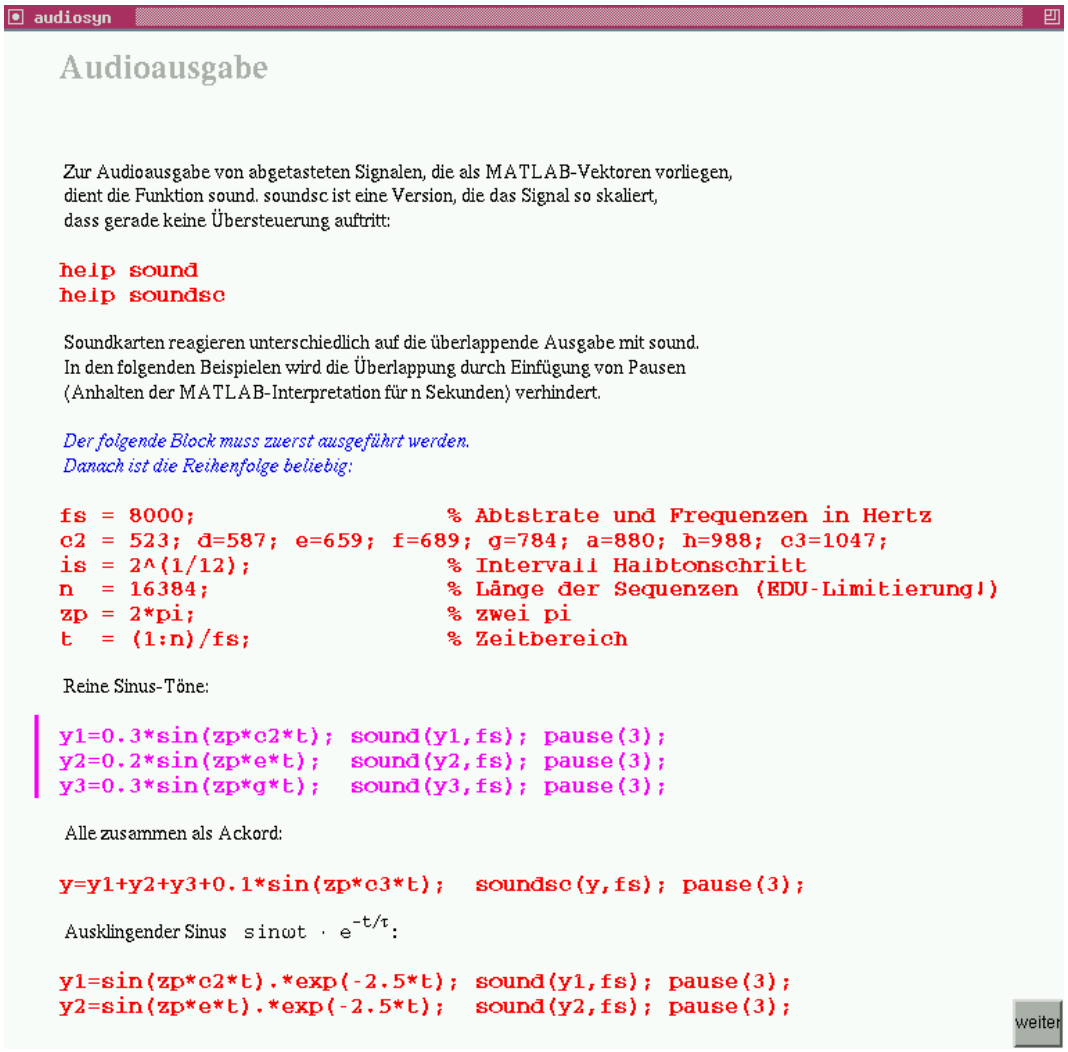


Abbildung 3: *Darstellung eines interaktiven Skripts im mscriptview-Browser am Beispiel Audiosynthese. Neben den üblichen Textformatierungen werden auch Formeln und Hyperlinks unterstützt, während Bilder in externen Fenster angezeigt werden. Die aktiven Codezeilen sind direkt in den Text eingebettet und werden farblich hervorgehoben. Beim Anklicken eines solchen Blocks wird der zugehörige Code ausgeführt: Im Beispiel werden Audiodaten berechnet und abgespielt.*

sondern auch für alle verbreiteten Varianten von Unix zur Verfügung (u.a. Linux, Solaris, MacOS X) und erlaubt damit den plattformunabhängigen Einsatz.

mscriptview

Zur Darstellung der Skripte dient *mscriptview*, ein selbstentwickelter Browser, der auf die in Matlab integrierten Funktionen zur Textformatierung zurückgreift und bei Bedarf modular erweitert werden kann. Die Grundidee besteht darin, die eigentlichen beschreibenden Texte der interaktiven Skripte als Kommentare direkt in die Matlab-Dateien mit den aktiven Codezeilen einzubetten. Auch Formeln lassen sich integrieren, da Matlab über Funktionen zur Darstellung von Symbolen und Formelzeichen in einer TEX-ähnlichen Syntax verfügt.

Auf diese Weise ergibt sich gleichzeitig eine sehr einfache Content-Erstellung, da alle thematisch zusammengehörenden Elemente in einer einzelnen Datei stehen und die Beschreibungen zusammen mit dem Code entwickelt werden können.

Problem der Lizenzkosten

Während sich die Kombination aus Matlab und mscriptview in Hauptstudiumsvorlesungen und -übungen seit mehreren Jahren sehr gut bewährt hat, stellen die Kosten und Lizenzbedingungen von Mathworks ein Hindernis für den breiten Einsatz im Grundstudium dar. Zwar verfügen sowohl der Fachbereich Informatik als auch das regionale Rechenzentrum der Universität Hamburg über eine Reihe von Matlab-Lizenzen, die von den Studierenden genutzt werden können. Die Anzahl der Lizenzen deckt aber keinesfalls die Teilnehmerzahlen in Grundstudiumsveranstaltungen ab. Zudem ist der Wunsch der Studierenden nachvollziehbar, die Software auch auf ihren eigenen Rechnern installieren zu wollen.

Student-Version

Leider kostet eine Vollversion der Matlab-Suite mit den grundlegenden Komponenten Matlab, Simulink und Symbolic Math Toolbox derzeit immerhin 1400 €, und für weitere wünschenswerte Komponenten wie die Signal-Processing oder Image-Processing Toolboxes entstehen zusätzliche Kosten. Im Rahmen der *Student Version* sind diese Komponenten bei vollem Funktionsumfang für einen deutlich reduzierten Preis von ca. 100 € erhältlich, wobei aber auch hier weitere Kosten für eventuell zusätzlich benötigte Toolboxes entstehen. Die mittlerweile nicht mehr erhältliche *Student Edition* basiert auf der älteren Version Matlab 5.3 und beinhaltet bereits die Signal Processing Toolbox, limitiert aber die Größe von Datenstrukturen auf maximal 32767 Elemente, was für Audio- und Bildverarbeitung eine empfindliche Einschränkung darstellt. Wegen der großen Verbreitung dieser Version werden die im Rahmen des Projekts entwickelten Skripte soweit wie möglich auch mit der Student Edition einsetzbar sein, um möglichst viele Anwender erreichen zu können.

2.2 Interaktive Skripte als HTML

Die in obigen Abschnitt skizzierte Softwarearchitektur basiert auf Matlab und dem zugehörigen *mscriptview*-Browser zum Anzeigen der Skripte. Gerade für Veranstaltungen im Hauptstudium hat sich diese Kombination sehr gut bewährt, zumal fast alle Studierenden wegen des hohen Nutzwerts eine eigene Matlab-Lizenz erwerben. Für den Einsatz im Grundstudium, aber auch für nicht technisch-mathematisch ausgerichtete Themengebiete, kann dies aber nicht vorausgesetzt werden. Damit ergibt sich ein Problem, denn bei Einsatz von *mscriptview* ist selbst zum passiven Durchblättern und Lesen der Skripte bereits eine vollständige Matlab-Installation notwendig.

HTML

Es stellt sich daher die Frage, ob und welche alternativen Plattformen zur Darstellung der Skripte genutzt werden können. Besonders attraktiv erscheint dabei die Umsetzung der interaktiven Skripte mittels HTML-Dateien. Dies ermöglicht die Darstellung der Skripte mit einem gewöhnlichen Web-Browser wie dem Internet Explorer oder Mozilla, wobei alle Zusatzfunktionen der Browser zur Verfügung

Vorteile...

stehen und der Anwender seine gewohnte Umgebung vorfindet. Auch die Integration in bestehende Web-Anwendungen oder E-Learning Frameworks wird deutlich erleichtert. Mehrere Vorteile der Verwendung eines HTML-Browsers sind offensichtlich:

- Die interaktiven Skripte können auch sauber angezeigt werden, wenn Matlab nicht zur Verfügung steht — dann allerdings passiv ohne die Interaktionsmöglichkeiten.

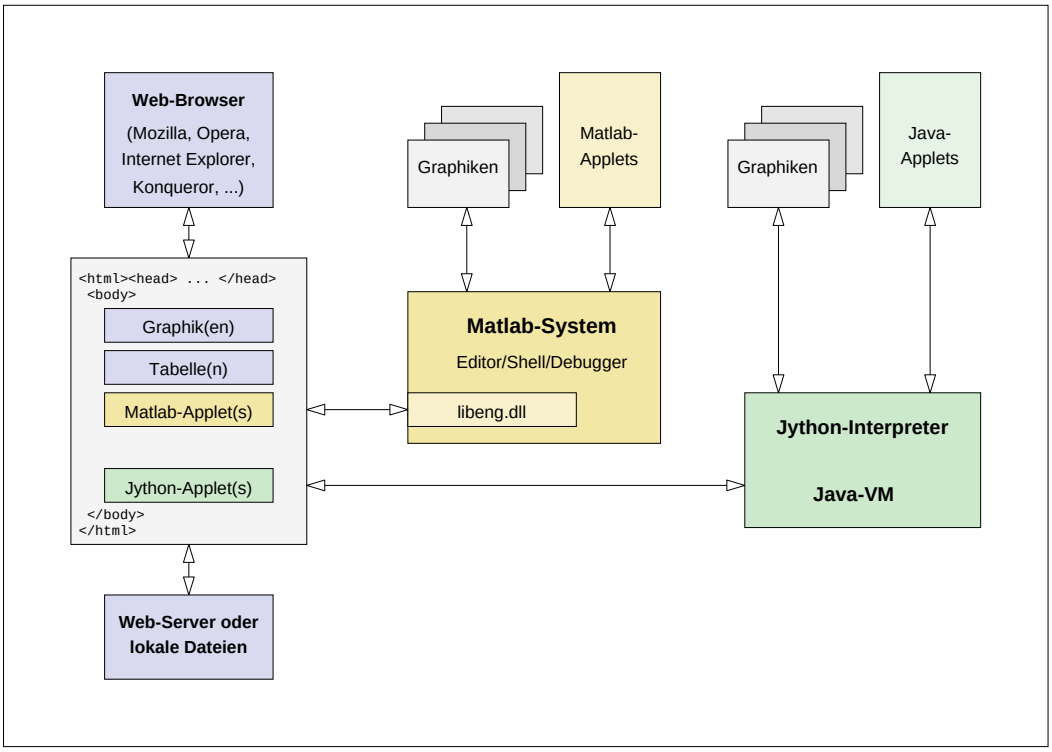


Abbildung 4: *Software-Architektur der HTML-basierten interaktiven Skripte. Die Inhalte werden als HTML-formatierte Webseiten aufbereitet und können mit jedem gewöhnlichen Browser angezeigt werden. Die aktiven Elemente sind als einfache Java-Applets realisiert und kommunizieren mit dem als externe Applikation laufenden Matlab-System bzw. dem Jython-Interpreter.*

- Bereitstellen der gewohnten Benutzeroberfläche inklusive der Voreinstellungen für Schriftarten, Schriftgrößen, Farben.
- Zugriff auf alle HTML- bzw. XHTML-Objekttypen wie eingebettete Abbildungen, Tabellen, Formulare und interaktive Objekte (Applets).
- Hyperlinks auf externe Webseiten und Ressourcen, Zugriff auf Suchmaschinen, komfortable Bookmarkverwaltung.
- Aufruf von Plugins oder externer Hilfsapplikationen wie etwa Postscript- oder PDF-Viewern.
- Möglichkeit zum Ausdrucken der Skripte inklusive aller Formatierungen und eingebetteten Graphiken.
- Integration in bestehende HTML-Plattformen oder E-Learning-Frameworks.
- Mögliche Integration in Content-Management Systeme und Nutzen bestehender HTML- oder XML-Editoren.
- Zugriff auf Sicherheitsfunktionen wie verschlüsselte Datenübertragung, Verwalten von Benutzerpasswörten, Überprüfung digital signierter Inhalte.

. . . und Nachteile

Auf der anderen Seite sind auch einige Nachteile zu verzeichnen. An erster Stelle steht dabei die teilweise deutlich aufwendigere Content-Erstellung im HTML-Format. Die Lösung besteht in der automatischen Umsetzung unserer vorhandenen Skripte vom mscriptview-Format nach HTML. Der zugehörige Konverter wird

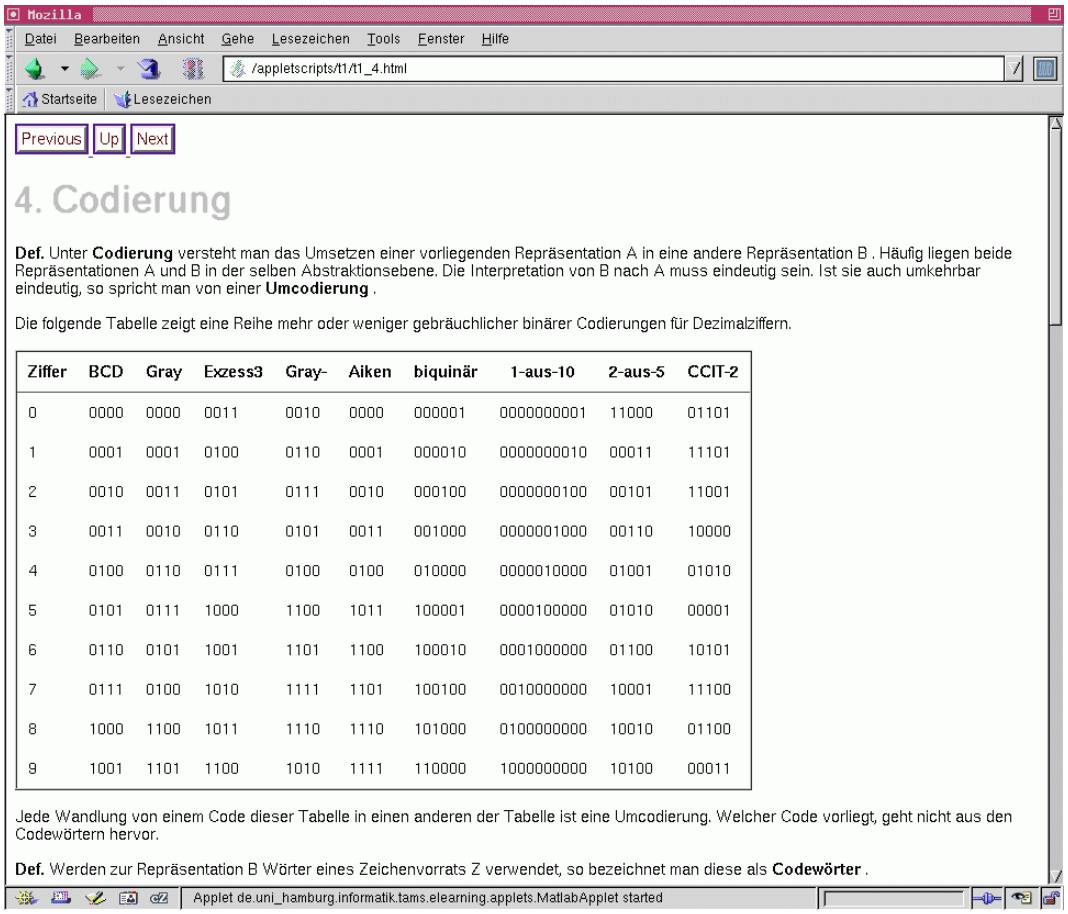


Abbildung 5: *Die HTML-Version der interaktiven Skripte stellt alle üblichen Konstrukte und Möglichkeiten von HTML zur Verfügung. Das Beispiel zeigt die erste Seite des Kapitels T1.4 (Codierung) aus der Vorlesung Technische Informatik 1 nach der der automatischen Umsetzung von Matlab nach HTML. Der Konverter erkennt die Dokumentstruktur inklusive der Absätze, erzeugt Hyperlinks zur Navigation, erkennt Formeln und Tabellen und übernimmt die Textformatierungen.*

in Abschnitt 2.3 vorgestellt. Ein besonders ärgerliches Problem bereitet die immer noch wenig standardkonforme Darstellung einiger HTML-Konstrukte durch die verschiedenen Browser. Zum Beispiel unterstützen alle getesteten Browser (u.a. Mozilla, Internet Explorer, Opera, Konqueror) nur (disjunkte) Teilmengen der in HTML 4 definierten mathematischen Symbole und Operatoren. Dies führt dazu, dass einige der Formelzeichen als Bilder bzw. Applets eingebunden werden müssen, obwohl sie eigentlich direkt in HTML definiert sind. Abhängig von Betriebssystemversionen und Druckertreibern können sich auch beim Ausdrucken Probleme ergeben.

Schließlich stehen in HTML nicht alle Funktionen von Matlab bzw. mscriptview zur Verfügung. Zum Beispiel stellt der *mscriptview*-Browser eine spezielle hierarchische Suchfunktion bereit, die in gängigen HTML-Browsern nicht verfügbar ist. (Natürlich könnte man die entsprechenden HTML-Seiten auch durch eine gewöhnliche Suchmaschine wie Google klassifizeren lassen oder auf serverbasierte Tools ausweichen).

Da alle aktuellen Browser zumindest HTML 4.0 bzw. XHTML inklusive CSS unterstützen, ist die Integration passiver Abbildungen, Tabellen, und der üblichen

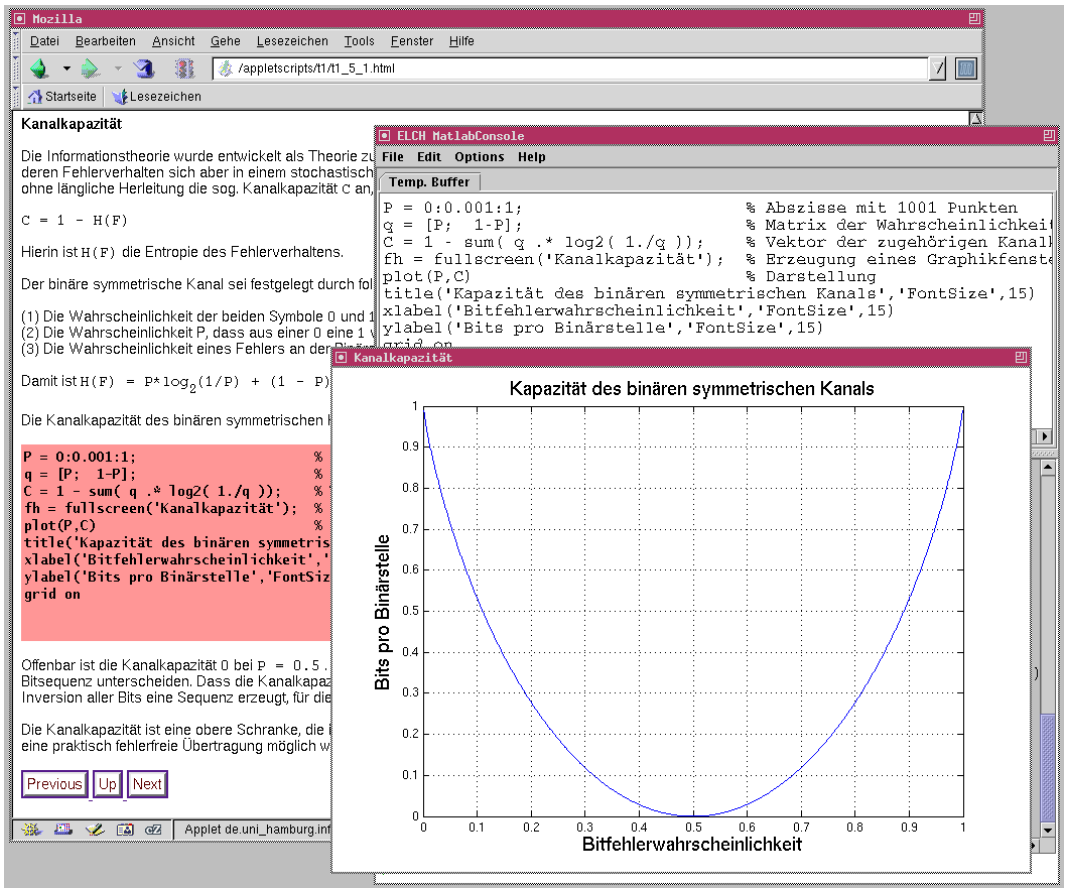


Abbildung 6: *In HTML werden die interaktiven Elemente mittels kleiner Java-Applets realisiert. Beim Anklicken eines solchen Applets wird der entsprechende Code an die zugehörige „Console“ (Matlab bzw. Jython) übergeben und dort ausgeführt. Dadurch stehen alle Funktionen der jeweiligen Plattform zur Verfügung; insbesondere auch die Plot-Funktionen mit zur Laufzeit berechneten Graphiken. Die Abbildung entstammt aus Kapitel T1.5.1 des T1-Skripts und demonstriert die Kanalkapazität des idealen, symmetrischen Kanals.*

HTML-Formulare überhaupt kein Problem. Spezielle Anforderungen an die Formatierung inklusive Blocksatz und pixelgenauer Anordnung lassen sich mit Style-Sheets realisieren. Damit bleibt für die konkrete Umsetzung nur noch die Frage üblich, wie sich die für das Konzept der interaktiven Skripte notwendigen Programmtexte integrieren lassen.

Interaktionskonzept

Obwohl das Interaktionskonzept zunächst nur auf dem einfachen Anklicken der eingebetteten Codeblöcke basiert, erscheint eine Realisierung mit JavaScript oder etwa als Flash-Plugin kaum möglich. Aus Gründen der Portabilität bleibt damit nur die Umsetzung mit Java-Applets, die direkt in die HTML-Seiten eingebettet werden. Die Applets übernehmen in diesem Konzept eine doppelte Funktion: erstens die Darstellung der interaktiven Elemente in den HTML-Seiten selbst und zweitens die Interaktion mit dem Benutzer und die Ausführung der ausgewählten und angeklickten Programme.

Wie die Übersicht in Abbildung 2 auf Seite 6 zeigt, sind dabei wiederum mehrere Varianten möglich. In der Variante b) dient das Applet nur als Vermittler und übergibt den ausgewählten Code an die darunterliegende Plattform zur Ausführung.

Beim Einsatz von Matlab bedeutet dies, dass weiterhin eine Matlab-Installation und Lizenz notwendig ist. Auch in Variante c) dienen die Applets nur als Vermittler und rufen externe Applikationen auf. Wie in Abschnitt 2.4 näher erläutert wird, lassen sich auf diese Weise insbesondere auch vorcompilierte Matlab-Funktionen aufrufen, für die keine Matlab-Lizenz erforderlich ist. In Variante d) schließlich übernimmt die Java-Umgebung direkt die Ausführung der ausgewählten interaktiven Elemente. Beispiele dafür werden in Abschnitt 2.5 vorgestellt.

2.3 Der MScript2Html-Konverter

Aufwendigere Content-Erstellung

Wie im letzten Abschnitt bereits angedeutet wurde, ist die Content-Erstellung für interaktive Skripte im HTML-Format gegenüber dem mscriptview-Konzept aufwendiger. Zwar unterstützt HTML praktisch alle für die Skripte notwendigen Textattribute sowie Graphiken und Hyperlinks; die aktiven Elemente müssen jedoch separat als Applets eingebunden werden. Leider unterstützen selbst gängige HTML-Editoren die Einbindung von Applets und ihrer Parameter nur unzureichend.

automatische Konvertierung

Als Lösung für dieses Problem wurde im Rahmen einer Diplomarbeit [35] ein Konverterprogramm entwickelt, dass die bestehenden Matlab-basierten Skripte für *mscriptview* automatisch nach HTML umsetzt. Damit kann die Content-Erstellung weiterhin auf ASCII-Basis mit jedem Texteditor erfolgen. Im einzelnen übernimmt der Konverter die folgenden Funktionen:

1. Einlesen der angegebenen Quelldateien und optional aller über Querverweise referenzierten Dateien, so dass die vollständige Umsetzung eines Skripts mit einem Aufruf möglich ist.
2. Analyse der Textstruktur und Trennung der aktiven Elemente (eingebetteter Matlab-Skriptcode) vom statischen Text.
3. Erzeugen von HTML-Code für Java-Applets für alle bei der Analyse erkannten aktiven Skripte. Der Skriptcode wird als Parameter für die Applets übergeben und von diesen angezeigt.
4. Umsetzung der statischen Texte nach HTML mit den passenden Formatierungen wie Formeln, Tabellen, Blocksatz, Fettschrift. Da die Formatierungen in den Matlab-Skripten nur teilweise explizit markiert sind, wird eine Heuristik zur Erkennung von Paragraphen und Tabellen eingesetzt.
5. Formeln werden ebenfalls nach HTML konvertiert, soweit die benötigten Symbole zur Verfügung stehen. Leider implementieren die verschiedenen Browser (z.B. Internet Explorer, Mozilla, Opera, Konqueror) jeweils nur einen Subset der vom W3C für HTML definierten Symbole. Komplexere Formeln können alternativ als Java-Applets oder als vorberechnete (z.B. via LaTeX erzeugte) Abbildungen eingebunden werden.
6. Erkennen von in den Skriptseiten eingebauten Hyperlinks und direkte Umsetzung in HTML-Links. Parallel dazu wird eine interne Liste aller entsprechenden Skriptseiten und Querverweise aufgebaut und zur Erzeugung von zusätzlichen Hyperlinks zur Navigation eingesetzt. Schließlich wird eine Indexseite aller Skriptseiten erzeugt, sofern diese nicht bereits im ursprünglichen Skript vorhanden war.

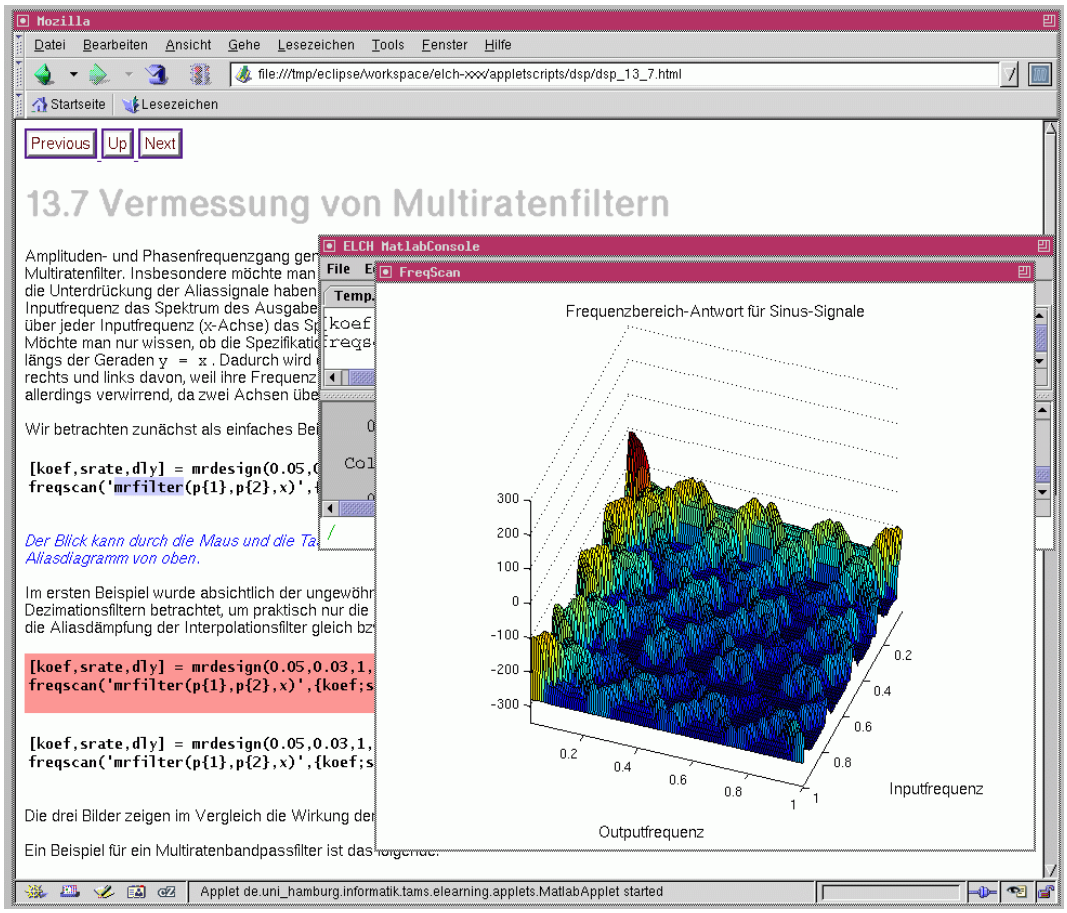


Abbildung 7: *Neben der Vorlesung Technische Informatik konnten auch weitere Vorlesungen bereits in das HTML+Applet Konzept integriert werden. Das Beispiel aus der Vorlesung Digitale Signalverarbeitung zeigt die Frequenzantwort eines Multiratenfilters als interaktiven 3D-Plot.*

Status

Der Konverter ist in Java geschrieben und kann wahlweise über die Kommandozeile oder mit einer Benutzeroberfläche gestartet werden. Die eingebauten Heuristiken wurden ursprünglich nur für die T1-Vorlesung entwickelt, mittlerweile aber mit allen vorhandenen Matlab-Dateien getestet. Sie erlauben die vollautomatische Umsetzung des Contents von Matlab nach HTML. Bis auf eine Handvoll Ausnahmen, aufgrund mehrdeutiger Formulierung oder besonders aufwendiger Formeln, sind keine manuellen Nacharbeiten notwendig. Damit ist auch die Integration unserer Materialien in die gängigen E-Learning Plattformen problemlos möglich. Derzeit stehen drei vollständige Vorlesungen (*Technische Informatik 1*, *Digitale Signalverarbeitung*, *Nachrichtentechnik und Datenübertragung*) zusätzlich zur Matlab-Version auch als HTML-Version zur Verfügung.

Die Abbildungen 5 bis 7 zeigen Beispiele für die Darstellung der derart umgesetzten Skripte in einem Standardbrowser (Mozilla). Zunächst zeigt Abbildung 5 die Umsetzung von beschreibendem Text und einer Tabelle. Die drei folgenden Abbildungen zeigen die Einbindung der interaktiven Elemente inklusive 2D- und 3D-Plots. Übrigens ermöglicht die HTML-Version auch das lineare Blättern im Skript (vorige und nächste Seite), während die zugrundeliegende Matlab-Version nur die direkte Navigation ausgehend vom Inhaltsverzeichnis vorsieht.

2.4 Matlab-Compiler

Trotz aller technischen Vorteile der Plattform Matlab und der bisherigen guten Akzeptanz beim Einsatz in Hauptstudiumsveranstaltungen bedeutet die Abhängigkeit von einer Matlab-Installation und -Lizenz ein echtes Problem für Lehrmaterialien, die auch im Grundstudium freizügig eingesetzt werden sollen. Zwar können die Studierenden die bestehenden Campus-Lizenzen der Universität für die Arbeit mit dem interaktiven Skript nutzen, allerdings nicht auf Ihren eigenen Rechnern sondern nur auf den Rechnern der Universität. Auch in Diskussionen mit anderen Lehrenden und ELCH Projektpartnern wurde immer wieder deutlich, dass die Lizenzkosten von Matlab als schwerwiegendes Hindernis aufgefasst werden.

Matlab-Compiler

In diesem Zusammenhang ergibt sich mit der im Juni 2004 erschienen neuen Version Matlab 7 (Release R14) eine interessante neue Option, die wir in den nächsten Wochen gründlich prüfen werden. Tatsächlich bietet die Firma Mathworks mit dem sogenannten *Matlab Compiler* schon länger ein Produkt an, mit dem aus vorhandenen Matlab-Funktionen standalone lauffähige Applikationen für Windows bzw. Linux erzeugt werden können. Die compilierten Applikationen umfassen alle benötigten Bibliotheken und dürfen bei Einhalten bestimmter Anforderungen ohne weitere Einschränkungen und Lizenzkosten weitergegeben werden.

Es liegt daher nahe, die interaktiven Skripte auf diese Weise zu compilieren und als fertige Applikationen an die Studierenden zu verteilen. Leider ließ sich diese Idee bisher nicht umsetzen, da der Matlab-Compiler in Versionen bis Matlab R13 nur eine kleine Teilmenge des gesamten Funktionsumfangs von Matlab selbst unterstützte. Die Einschränkungen betrafen auch fundamentale und für das interaktive Skript unverzichtbare Funktionen wie die Graphikschnittstelle und die Dateiein- und ausgabe.

Matlab R14

Die Situation hat sich allerdings mit Einführung von Matlab R14 deutlich verbessert, da die aktuelle Version des Compilers jetzt fast den kompletten Funktionsumfang von Matlab umfasst (mit Ausnahme der Java-Schnittstelle und einiger Toolboxes). Allerdings sind Zugriffe auf den Matlab-Workspace und bestimmte Funktionen weiterhin nicht möglich bzw. durch die Lizenzbedingungen ausdrücklich untersagt. Dies ist verständlich, da ansonsten letztlich der volle Funktionsumfang von Matlab zur Verfügung stände.

Einschränkungen

Die fehlende Java-Schnittstelle ist dabei trotz der in unseren Skripten eingesetzten Java-Applets leicht zu verschmerzen, da diese Applets auch ohne Matlab lauffähig sind. Dafür bedeutet der fehlende Zugriff auf den Matlab-Workspace eine Einschränkung, die sich vermutlich nur durch eine vollständige Änderung der Softwarearchitektur umgehen lässt. Das Problem besteht darin, dass der *mscriptview*-Browser die auszuführenden Skripte erst zur Laufzeit (nach Anklicken durch den Anwender) aus Texten zu Matlab-Code zusammenbaut und dann vom Matlab-Workspace ausführen lässt. Diese Funktion steht für die vorcompilierten Anwendungen aber weiterhin nicht zur Verfügung.

Ein möglicher Ausweg ist die Kombination von vorcompilierten Matlab-Funktionen mit der im letzten Abschnitt vorgestellten HTML+Applet-Architektur. In diesem Fall dienen die Applets weiterhin zur Präsentation der aktiven Elemente und zur Interaktion mit dem Anwender. Anstatt die angeklickten Skripte aber vom Matlab-Workspace ausführen zu lassen, werden statt dessen die passenden vorcompilierten

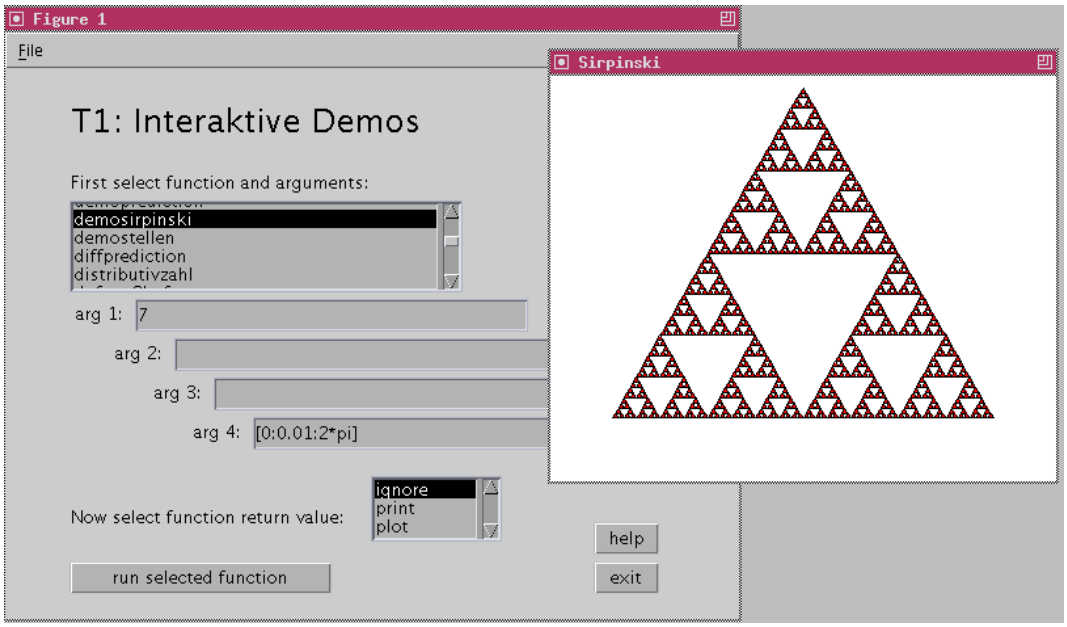


Abbildung 8: *Hilfsprogramm zum Aufruf der vorcompilierten Matlab Programme. Nach Auswahl der gewünschten Funktion können die gewünschten Funktionsparameter eingestellt werden (hier z.B. die Rekursionstiefe von 7 für das Sirpinski-Dreieck), so dass exploratives Lernen in vollem Umfang unterstützt wird.*

Applikationen direkt aufgerufen. Dieser Ansatz erscheint so attraktiv, dass wir derzeit entsprechende Experimente mit einer von Mathworks kurzfristig zur Verfügung gestellten Testversion des Compilers durchführen. Damit erscheint es jetzt erstmals möglich zu sein, eine leicht eingeschränkte Version der interaktiven Skripte zu erstellen, die sich ohne Lizenzprobleme und -kosten verteilen lässt. Der volle Funktionsumfang bleibt allerdings weiterhin der bestehenden Version vorbehalten, die beim Anwender eine komplette Matlab-Installation voraussetzt.

Da es sehr aufwendig wäre, zu jeder unserer vorhandenen (ca. 1000) Matlab-Funktionen eine einzelne Applikation zur Übergabe der Funktionsparameter und Auswertung der Rückgabewerte zu erstellen, werden wir eine Hilfsapplikation verwenden, die diese Aufgaben übernimmt. Ein erster Prototyp dieses Werkzeugs ist in Abbildung 8 dargestellt. Nach Auswahl der aufzurufenden Funktion können die gewünschten Parameter und die Anzeige der Rückgabewerte eingestellt werden. Derzeit sind bis zu vier Argumente vorgesehen, die in Matlab-Notation eingegeben werden, so dass auch komplexe Ausdrücke wie Matrizen, Boole'sche Ausdrücke oder arithmetische Funktionen übergeben werden können. Eigene Experimente der Studierenden sind daher weiterhin in vollem Umfang möglich; nur die Auswahl der zur Verfügung stehenden Funktionen ist eingeschränkt.

2.5 HTML-Skript mit Jython-Applets

Auch in der vierten Variante unserer interaktiven Skripte dienen in die HTML-Seiten eingebettete Java-Applets zur Anzeige der aktiven Codeblöcke und zur Interaktion mit dem Benutzer. Gleichzeitig übernimmt die Java Laufzeitumgebung der Applets aber auch die Ausführung der ausgewählten Codeblöcke. Für den Anwender hat dieser Ansatz den Vorteil, dass keine zusätzliche Softwareinstallation

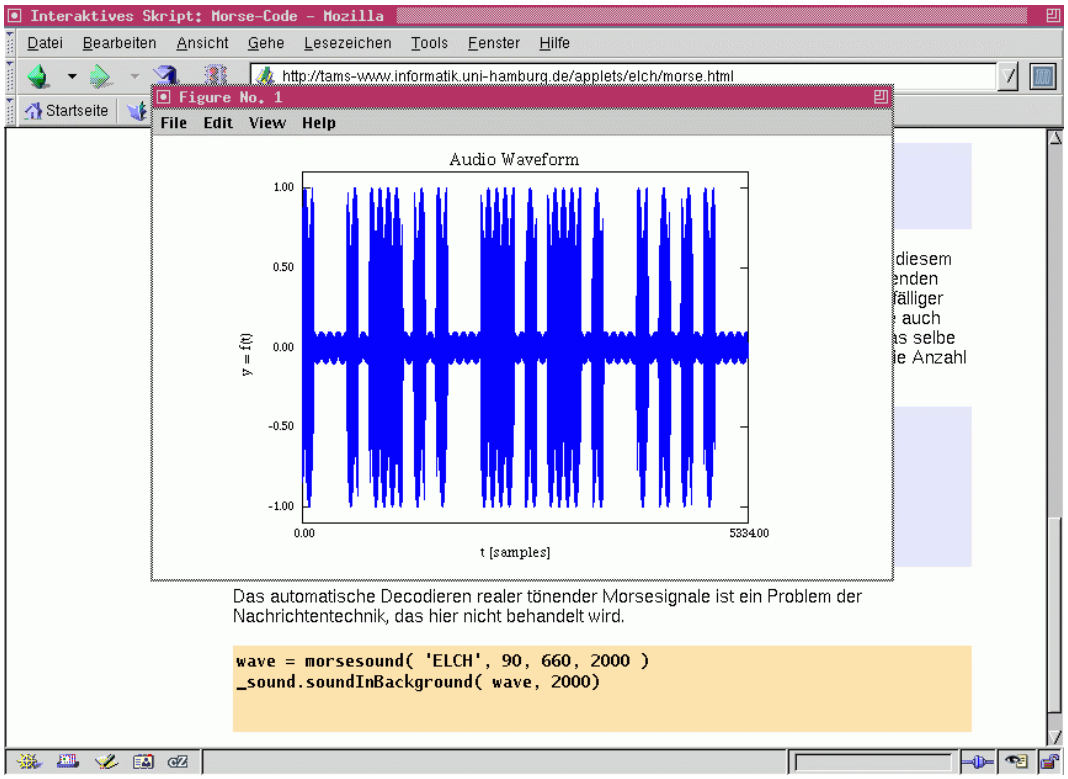


Abbildung 9: *Alternative Plattform: die Jython/HTML-Version des interaktiven Skripts verwendet HTML als Dokumentenformat und kann mit jedem Java-fähigen Browser angezeigt werden. Anstelle von Matlab dienen Jython-Skripte als interaktive Elemente. Im Beispiel wird der eingegebene Textstring zunächst im Morsecode kodiert, daraus ein Audioclip via Amplitudenmodulation erstellt, und dieser dann geplottet und abgespielt.*

erforderlich ist, da alle von den Applets benötigten Java-Klassen und weitere Dateien automatisch vom Webserver nachgeladen werden.

Jython

Wie bereits im ersten Projektbericht ausführlich erläutert wurde [11], stellt die übliche Java Laufzeitumgebung nicht den vollen für die interaktiven Skripte benötigten Funktionsumfang zur Verfügung. Insbesondere fehlt die Möglichkeit, zur Laufzeit interaktiv den Programmcode zu ändern und diese Änderungen sofort auszuprobieren. Als Abhilfe setzen wir auf die Kombination der Java Laufzeitumgebung mit dem Jython Interpreter, der genau diese Möglichkeiten nachrüstet und auch in Applets problemlos eingesetzt werden kann.

Downloadzeiten

Zusammen mit einer für das Projekt entwickelten Benutzeroberfläche mit Editor und Protokollfenstern umfasst die gesamte Software etwas über 1 MByte an Programmcode. Bei Webzugriff über das Netzwerk des Fachbereichs Informatik bedeutet dies Downloadzeiten von unter einer Sekunde, und auch für DSL-Verbindungen ergeben sich unkritische Wartezeiten von etwa 10 Sekunden. Bei ISDN (64kb/s) und Modemverbindungen (40kb/s) ist dagegen mit Downloadzeiten von bis zu 4 Minuten zu rechnen. Da dieser Download aber nur beim ersten Starten des Applets anfällt, während spätere Aufrufe direkt aus dem Browsercache geladen werden, erscheint uns diese Wartezeit gerade noch akzeptabel. Alternativ ist natürlich die lokale Installation der gesamten Software (HTML-Dateien und Applet-Klassen) beim Anwender möglich. Wir haben die Software bereits für den Einsatz des

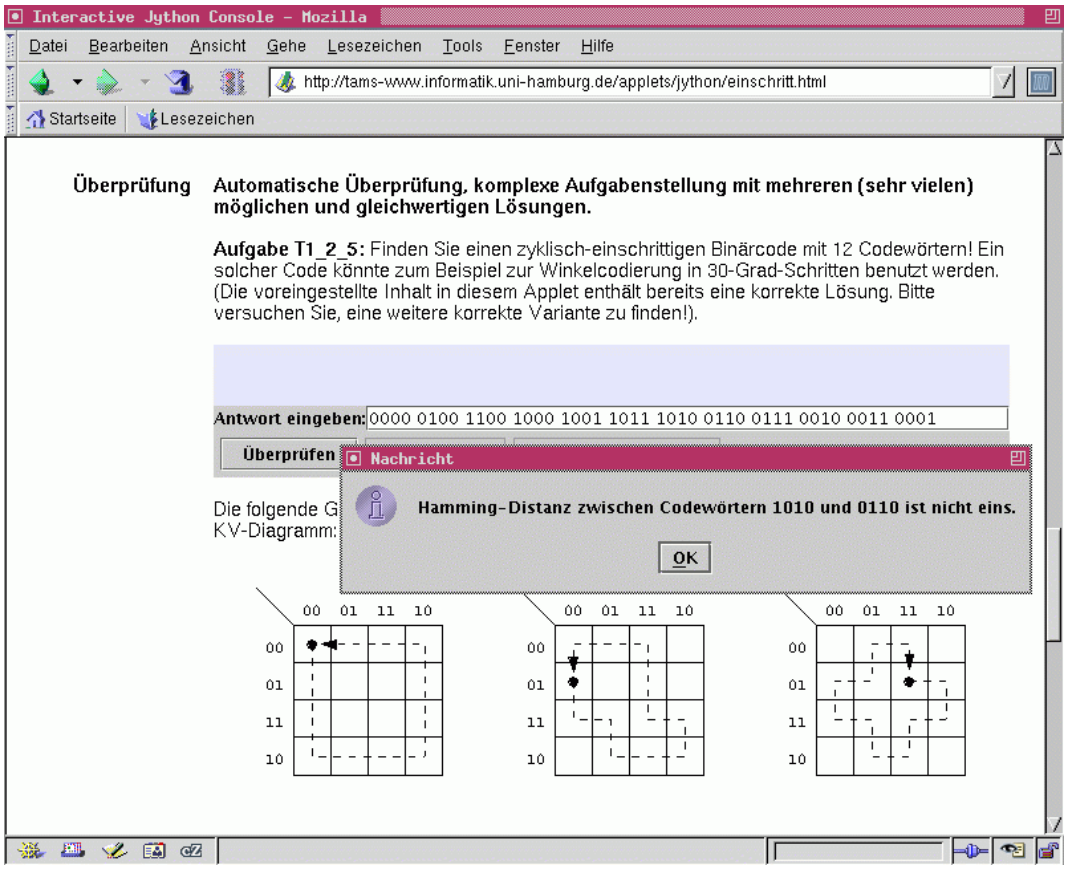


Abbildung 10: *Überprüfung der eingegebenen Lösung zu einer Übungsaufgabe und kontextabhängige Hilfestellung mittels Java-Applet. Vorteil dieser Variante des interaktiven Skripts ist, dass alle benötigte Software vom Webserver geladen wird und daher keine zusätzliche Softwareinstallation notwendig ist.*

Java-Webstart Protokolls [25] vorbereitet, so dass sich der Vorgang des einmaligen Downloads und die anschließende lokale Installation automatisieren lässt. In dieser Variante ist es möglich, die Codegröße der Applets in den HTML-Seiten auf wenige KBytes zu reduzieren, so dass selbst über Modemverbindungen keine Verzögerungen auftreten.

Integration von Übungsaufgaben

Wegen des einfachen Zugriffs auf die Jython-Applets ohne zusätzliche Softwareinstallation wurden die meisten Alorithmen zur Überprüfung von Übungsaufgaben auf diese Weise implementiert. Je nach Aufgabenstellung können die Algorithmen dabei rein als Java-Klassen oder als Kombination von compilierten Java-Klassen mit interpretierten Jython-Skripten realisiert werden. Abbildung 10 zeigt die interaktive Überprüfung der bereits in den vorangegangen Projektberichten als Beispiel herangezogenen Übungsaufgabe T1.2.5. In der Abbildung erkennt die Überprüfung einen Fehler in der eingegebenen Lösung und antwortet mit einer sinnvollen Hilfestellung.

Die Jython-Applet Version der im letzten Projektbericht [11] beschriebenen Signaturanalyse zur Überprüfung digitaler Schaltungen ist in Abbildung 11 dargestellt. Während des Schaltungsentwurfs kann zusätzlich zur interaktiven Simulation jederzeit auch die Überprüfung durch das Jython-Applet gestartet werden. Im gezeigten Ausschnitt aus der gesamten Schaltung (4-bit Binärzähler mit Enable und Carry-Out) ist deutlich zu erkennen, dass noch ein Gatter fehlt. Entsprechend liefert die

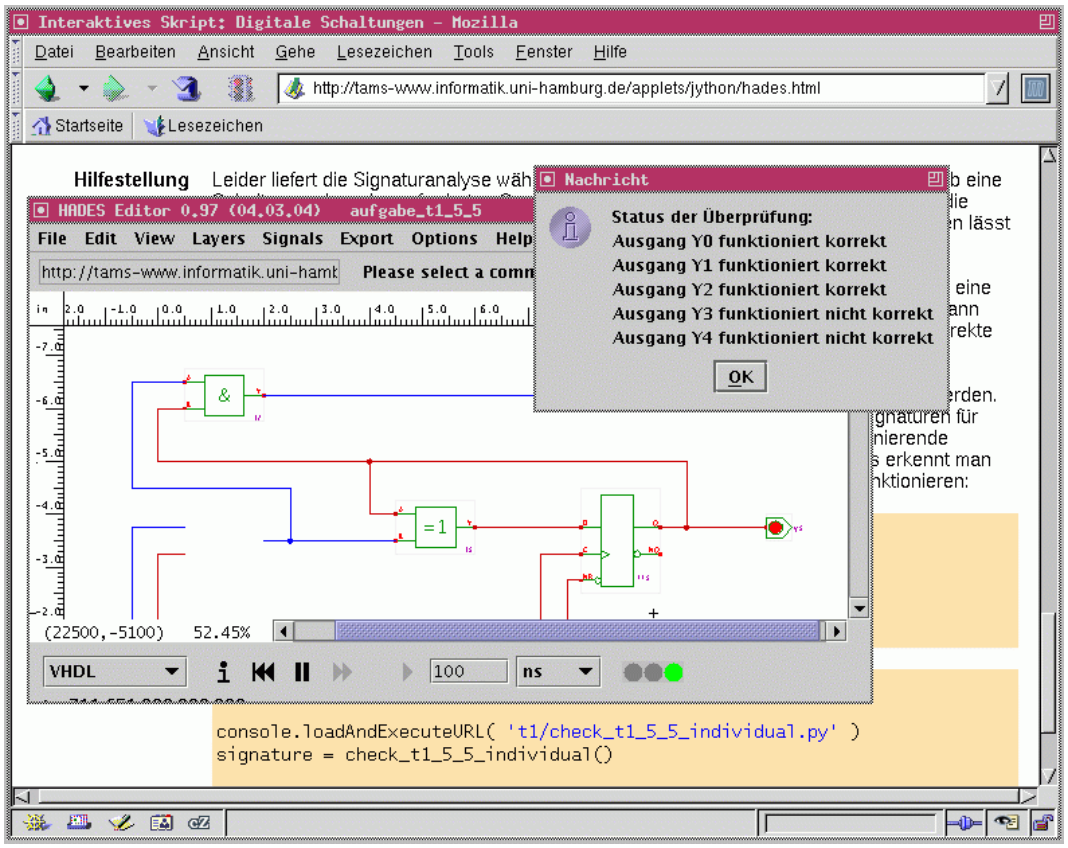


Abbildung 11: *Jython-Applet zur Überprüfung einer digitalen Schaltung via Signaturanalyse (für Details der eingesetzten Algorithmen siehe [11]). In der Abbildung fehlt noch ein UND-Gatter des zu entwerfenden 4-bit Binärzählers, so dass die Ausgänge Y3 und Y4 falsche Werte liefern.*

Signaturanalyse den gezeigten Hinweis (Ausgänge Y0..Y2 funktionieren korrekt, Ausgänge Y3 und Y4 nicht), der die Fehlersuche bereits auf einen kleinen Teil der gesamten Schaltung einschränkt.

Eine Reihe weiterer Demonstrations-Applets steht unter der URL *http://tams-www.informatik.uni-hamburg.de/applets/jython/* zum Ausprobieren zur Verfügung. Die Liste der verfügbaren Applets wird während des WS 04/05 um die jeweils aktuellen Übungsaufgaben zur Vorlesung Technische Informatik 1 ergänzt.

3 Überprüfung von Formeln

Wie bereits im ersten Projektbericht ausführlich erläutert wurde, nimmt die Kategorie der *Formel-Aufgaben* mit ihren Unterklassen einen breiten Raum in der technischen Informatik ein (vgl. Abbildung 1 auf Seite 3). Die in diesem Kapitel vorgestellten Strategien zur Überprüfung dieser Art von Aufgaben dürften sich aber auch für die meisten anderen naturwissenschaftlichen und ingenieurwissenschaftlichen Fächer direkt übernehmen lassen.

Übersicht

Zunächst präsentiert Abschnitt 3.1 einige typische Beispiele für die verschiedenen Varianten von Aufgabenstellungen und der als Lösung erwarteten Formeln. Daraus werden in Abschnitt 3.2 einige Konzepte zur Überprüfung dieser Aufgabenstellungen abgeleitet.

Einen „direkten“ Zugang zur Überprüfung von Formeln bieten Programmsysteme zur symbolischen Mathematik. Abschnitt 3.3 liefert eine Einführung in die Thematik und skizziert Implementierungsstrategien am Beispiel der Matlab Symbolic Math Toolbox. Als Alternative kommt die numerische Auswertung der Formeln in Frage; die dabei zu berücksichtigenden Anforderungen werden in Abschnitt 3.4 zusammengestellt.

Wegen der besonderen Bedeutung für die technische Informatik verdient die Überprüfung von Boole’schen Ausdrücken ihren eigenen Abschnitt 3.5. Da die in den Übungsaufgaben vorkommenden Funktionen nur wenige Variablen umfassen, hat sich die Umwandlung in Normalformen bzw. Funktionstabellen trotz der theoretischen Komplexitätsprobleme als praxistauglich erwiesen und ermöglicht auch sinnvolle Hilfestellungen. Zusätzlich werden auch Nebenbedinungen wie etwa die Realisierung nur mit NAND-Gattern und graphische Verfahren wie die bekannten KV-Diagramme unterstützt.

3.1 Beispiele

Als Einführung in die Problematik der *Formel-Aufgaben* werden in diesem Abschnitt exemplarisch einige Formeln gezeigt, die sich als Lösung zu Übungsaufgaben der Vorlesung Technische Informatik (Teile 1 und 2) bzw. des Übungsbuches [24] ergeben. Falls die Auswertung der Formel(n) einen einfachen Zahlenwert ergibt, ist dieser mit angegeben:

- T1.3.1a (mittlerer Informationsgehalt):
  $H = \sum_{i=1}^{N} p(x_i) \cdot \mathrm{ld}\frac{1}{p(x_i)} \approx 3.32\,bit$
- T1.3.5 (Gleichungssystem über GF2, die Lösung umfasst 3 Gleichungen):
  $x+y=1, y+z=0, x+y+z=1$
- T1.5.1a (Zeitverhalten):
  $y(t) = b(t-2\tau) \vee \neg b(t-3\tau)$
- T1.5.5 (Binärzähler, vier Gleichungen):

$$D_0 = C_e \oplus z_0$$
$$D_1 = C_e z_0 \oplus z_1$$
$$D_2 = C_e z_0 z_1 \oplus z_2$$
$$D_3 = C_e z_0 z_1 z_2 \oplus z_3$$

- Schiffmann-Schmitz Aufgabe 2 (Elektronenstrahlröhre):
  $Y = y + y' = \frac{1}{4}\frac{U}{d}\frac{l^2}{U_A} + L \cdot \frac{U}{d} \cdot \frac{l}{2U_A}$
- Aufgabe 5 (Widerstandsnetzwerk):
  $R_{23} = R_2 || R_3 = \frac{R_2 \cdot R_3}{R_2 + R_3} \approx 6.6 k\Omega$
- Aufgabe 14 (Effektivwert):
  $U_{1,\text{eff}} = \sqrt{\frac{1}{T}\int_0^T (\hat{u} \cdot \sin(\frac{2\pi}{T}t))^2 \mathrm{d}t}$
- Aufgabe 17 (Ladevorgang am Kondensator):
  $u_c(t) = U_0(1 - e^{-t/\tau})$

3.2 Konzepte zur Überprüfung

Obwohl die obigen Beispiele nur einen winzigen Ausschnitt aus dem Spektrum der möglichen Anwendungsfälle zeigen, wird doch bereits deutlich, dass abhängig von der jeweiligen Aufgabenstellung unterschiedliche Strategien zur Überprüfung besonders geeignet oder sogar notwendig sein werden.

Eingabeformate

Der erste Schritt betrifft offenbar die Definition eines Eingabeformats für die Überprüfung. Wie ebenfalls bereits im ersten Projektbericht erläutert wurde, kommt das automatische Einscannen handgeschriebener Formeln wegen des hohen Aufwands und der geringen Erkennungsrate nicht in Frage. Auch auf die Formeleditoren der gängigen Textverarbeitungen muss leider verzichtet werden, da die zugrundeliegenden Dateiformate unzureichend dokumentiert sind und keine brauchbaren Konverter zur Interpretation solcher Formeln verfügbar sind.

Auf der anderen Seite sind die meisten der für Formeln benötigten Symbole nicht im Standard-ASCII Zeichensatz enthalten. Damit bleibt nur die Eingabe der Formeln als Unicode-formatierter Text oder als erweiterter ASCII-Text mit vorher vereinbarten Symbolen. Für letztere Variante eignet sich insbesondere die weitverbreitete Syntax des TEX-Satzsystems, die (mit Einschränkungen) von den meisten Programmsystemen für symbolische Mathematik unterstützt wird. In jedem Fall wird es notwendig sein, neben den Sub- und Superskripten für Variablennamen und den griechischen Buchstaben auch die elementaren Funktionen (etwa `\sin(x)`) sowie die Eingabe von Summen, Integralen, Wurzeln etc. zu erlauben.

Symbolische Manipulation

Die mit Papier und Bleistift übliche Transformation von Gleichungen durch symbolische Manipulation erlaubt die Überprüfung im Prinzip durch Berechnung der Differenz der eingegebenen Gleichung $f(x)$ und der zugehörigen Musterlösung $\hat{f}(x)$. Offenbar ist die Lösung korrekt, wenn $f(x) - \hat{f}(x) \equiv 0$. Dieser Ansatz wird in Abschnitt 3.3 am Beispiel der *Symbolic Math Toolbox* von Matlab erläutert. Leider gibt es trotz des hohen Entwicklungsstandes der Werkzeuge zur symbolischen Mathematik keinerlei Garantien dafür, dass die symbolische Vereinfachung der Eingabe

gelingt, zumal auch sehr komplexe oder ungewöhnliche Ausdrücke zu berücksichtigen sind. Eine Ausnahme ergibt sich, falls für die Aufgabenstellung eine eindeutige Normalform existiert.

Numerische Auswertung

Die Alternative zur symbolischen Manipulation ist die direkte numerische Auswertung von Gleichungen und Funktionen. Aus Gründen des Rechenaufwands sollte die Anzahl der auszuwertenden Eingangswerte natürlich möglichst gering sein; entsprechend gibt es verschiedene Varianten zur Auswahl geeigneter Eingangswerte inklusive der zufälligen Auswahl. Nur für Funktionen über diskreten Wertemengen ist es möglich, alle einzelnen Eingangswerte auszuprobieren und damit eine vollständige Überprüfung sicherzustellen. Die für die technische Informatik besonders wichtige Anwendung auf Boole'sche Funktionen wird in Abschnitt 3.5 beschrieben.

Die folgende Liste fasst noch einmal diese prinzipiellen Möglichkeiten zusammen:

- Symbolische Manipulation und Vereinfachung der Gleichungen, anschließend Vergleich mit einer Musterlösung oder Überprüfung von geeignet vorgegebenen Bedingungen (z.B. Lage von Nullstellen und Maxima).
- Symbolische Manipulation durch Umwandlung der Gleichungen in eine Normalform und anschließender Vergleich auf Identität. Dieses Verfahren kann nur eingesetzt werden, wenn eine Normalform existiert (z.B. für logische Ausdrücke oder Polynome).
- Vollständige numerische Auswertung der Gleichungen für alle möglichen Parameterwerte. Dieser Ansatz ist natürlich nur für Funktionen über diskreten Wertemengen möglich (z.B. Boole'schen Ausdrücken).
- Numerische Auswertung der Gleichungen für geeignet ausgewählte Parameterwerte und Vergleich mit den zugehörigen Werten einer Musterlösung, unter Berücksichtigung von Abweichungen (z.B. durch unterschiedliche Modellierung oder Rundungsfehler).
- Numerische Auswertung der Gleichungen für zufällig ausgewählte Parameterwerte (Monte-Carlo-Verfahren) und Vergleich mit den zugehörigen Werten der Musterlösung.
- Berücksichtigung möglicher Abweichungen der Lösungsansätze von der Musterlösung durch unterschiedliche Modellierung, unterschiedliche Abschätzungen von Parametern, und Rundungsfehler.
- Kombination und Schachtelung mehrerer Verfahren.
- Ergänzung um Plausibilitätstests, um einige typische Fehler (z.B. Vorzeichenfehler) besser erkennen und für sinnvolle Hilfestellungen bzw. Fehlermeldungen nutzen zu können.

3.3 Symbolic Math Toolbox

Direkter Stringvergleich: nicht sinnvoll

Trotz der Forderung nach einfacher ASCII-Formatierung der Eingabe für die Überprüfung von Formeln ist offensichtlich, dass ein einfacher Textvergleich mit einer Musterlösung nur in den wenigsten Fällen in Frage kommt. Auf diese Weise wäre zwar eine der Musterlösung exakt entsprechende Eingabe zu erkennen, aber bereits die einfache Umstellung der Terme oder Variablen würde den Textvergleich überfordern. Auch der Versuch, die Lösung mit einfachen Stringmanipulationen (Löschen von Leerzeichen, Sortieren von Termen, etc.) in die Musterlösung zu überführen, ist angesichts der Vielfalt der möglichen gleichwertigen Schreibweisen für ein und dieselbe Funktion zum Scheitern verurteilt.

Symbolische Manipulation

Als Alternative liegt es nahe, sich auf die mittlerweile sehr leistungsfähigen Programmsysteme für *symbolische Mathematik* zu verlassen, die ihrerseits ausgefeilte Algorithmen für die symbolische Manipulation von Funktionen enthalten. Drei bekannte Softwarepakete dieser Kategorie sind Mathematica, Maple und MuPAD. Die als optionale Komponente für Matlab verfügbare *Symbolic Math Toolbox* basiert auf dem Programmkern von Maple und integriert dessen Algorithmen in den Matlab-Workspace. Über eine Reihe von Hilfsfunktionen können symbolische Ausdrücke auch mit numerischen Algorithmen kombiniert werden; unter anderem ist es möglich, Ausdrücke sowohl symbolisch, numerisch mit beliebiger Genauigkeit („variable precision arithmetic“) oder der gewöhnlichen Gleitkommaarithmetik auszuwerten.

Die Grundidee für die Überprüfung eines eingegebenen Ausdrucks $f(x)$ mit einem solchen Programmsystem ist der direkte Vergleich mit der zugehörigen Musterlösung $\hat{f}(x)$. Sofern die Berechnung der Hilfsgröße $\Delta f = f(x) - \hat{f}(x)$ exakt den Wert Null ergibt, sind die beiden Ausdrücke identisch und die Lösung ist korrekt. Falls Δf nicht Null ist, dient der resultierende Ausdruck direkt als Hilfestellung. Der entscheidende Unterschied gegenüber dem einfachen Stringvergleich besteht darin, dass die Programmpakete automatisch eine ganze Reihe von Heuristiken zur Minimierung der eingegeben Ausdrücke verwenden. Dabei ist die Leistungsfähigkeit der eingesetzten Heuristiken erstaunlich hoch; zum Beispiel erkennt Maple die Additionstheoreme für die Winkelfunktionen wie $\sin^2(x) + \cos^2(x) = 1$ und nutzt diese für die Vereinfachung aus:

```
syms omega t;                       % symbolische Variablen

f = 1 - cos(t)^2;                   % entspricht sin(t)^2
F = sin(t)^2;                       %

disp( f - F );                      % direkte Auswertung
    1-cos(t)^2-sin(t)^2             % Rohdaten

disp( simple(f - F^2)):             % Vereinfachung
    0                               % ok.
```

Das obige Beispiel demonstriert nicht nur das Konzept, sondern auch den prinzipbedingten Nachteil: es gibt keine Garantie, dass die eingesetzten Heuristiken für

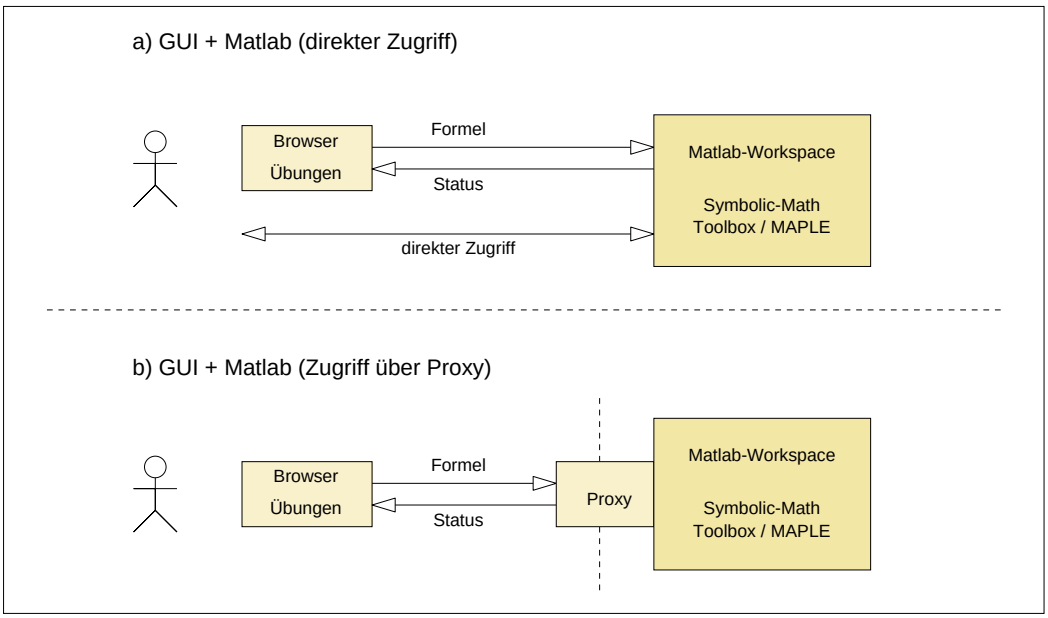


Abbildung 12: *Direkter und indirekter Zugriff auf die Funktionen der Matlab Symbolic-Math Toolbox. Letzteres kann notwendig sein, um ein direktes Ablesen der Musterlösungen zu verhindern.*

eine gegebene Funktion und Musterlösung tatsächlich die notwendigen Umformungen zum Nachweis der Gleichheit finden. Nur für wenige Formeln (etwa Polynome oder die in Abschnitt 3.5 beschriebenen Boole'schen Ausdrücke) existieren eindeutige Darstellungen. Um eine eigentlich korrekte Lösung nicht fälschlich zurückzuweisen, kann die reine symbolische Umwandlung deshalb in einem mehrstufigen Prozess um weitere Schritte ergänzt werden. Bei der Symbolic Math Toolbox ist es zum Beispiel möglich, die verschiedenen Algorithmen zur Vereinfachung eines Ausdrucks auch direkt aufzurufen (Funktionen `simple()`, `simplify()`, `factor()`, usw.) und auf diese Weise weitere Varianten der Vereinfachung durchzuprobieren.

Maximumsuche

Im nächsten Schritt ist es möglich, immer noch mit symbolischen Verfahren das Maximum der Funktion $|f - \hat{f}|$ zu suchen. Dieser Schritt ist auch notwendig, falls die Aufgabenstellung eine gewisse (absolute oder relative) Abweichung σ zwischen der eingegebenen Lösung und der vorgegebenen Musterlösung zulässt. In diesem Fall wird die Lösung akzeptiert, sofern $\max_x |f(x) - \hat{f}(x)| \leq \sigma$. Falls die symbolische Vereinfachung ein geeignetes Maximum findet, erlaubt dies sofort eine Aussage über die Korrektheit (und Qualität) der aktuellen Lösung. Im letzten Schritt ist es möglich, die Überprüfung zusätzlich um die numerische Auswertung von $f(x) - \hat{f}(x)$ zu ergänzen, vgl. Abschnitt 3.4.

Rundungsfehler

Schließlich erlaubt die symbolische Verarbeitung in vielen Fällen auch die Kontrolle der Rundungsfehler, die bei der numerischen Rechnung mit Gleitkommaarithmetik unvermeidlich sind. Ein besonders eindrucksvolles Beispiel ergibt sich mit den Variablenwerten $x = 10864$, $y = 18817$ für die folgende Formel $9 \cdot x^4 - y^4 + 2 * y^2 = 1$. Die Auswertung im Rahmen der gewöhnlichen IEEE-754 Gleitkommaarithmetik liefert den Integerwert 2, während die Berechnung mit ausreichender Genauigkeit den mathematisch korrekten Wert 1 liefert. Obwohl das Resultat der Gleitkommarechnung als Integerzahl erscheint und daher nichts auf Rundungsfehler hindeutet, ist also nicht einmal die erste Stelle des Ergebnisses korrekt.

Im Kontext des interaktiven Skripts ist übrigens wieder zu berücksichtigen, dass die Anwender jederzeit auch eigene Befehle in der interaktiven Programmumgebung eingeben können und außerdem Zugang zu den Quelltexten der einzelnen Funktionen haben. Im Beispiel von Seite 22 kann ein Student sich die gesuchte Lösung also einfach durch Eingabe des Befehls `disp(simple(F))` oder `disp(pretty(F))` sauber formatiert anzeigen lassen. Falls dies nicht gewünscht ist, bleibt nur die Kapselung der interaktiven Programmierumgebung hinter einer geeigneten Software, etwa einem Web-Proxy (vgl. Abbildung 12).

3.4 Numerische Auswertung

Einzige Voraussetzung für die numerische Auswertung von Formeln ist eine Plattform, die als Text eingegebene Ausdrücke zur Laufzeit auswerten kann. In Matlab leisten dies die Funktionen `eval()` und `feval()`, in Jython ebenfalls die Funktion `eval()`. Natürlich bietet Matlab in dieser Hinsicht einen deutlich größeren Funktionsumfang, da die Ausdrücke von vornherein auch mit Arrays bzw. Matrizen umgehen können. Bis auf die syntaxbedingten Einschränkungen können alle Beispiele aus Abschnitt 3.1 problemlos in Matlab-Syntax eingegeben werden; nur die als Bezeichner nicht zulässigen griechischen Buchstaben müssen durch passende Namen ersetzt werden. Jython dagegen unterstützt zunächst nur Operationen auf Skalaren und nur die elementaren mathematischen Funktionen. Es existiert allerdings eine Erweiterung, mit der eine weitgehend Matlab-kompatible Schreibweise von Array-Operationen nachgerüstet werden kann. In beiden Plattformen lassen sich zusätzlich benötigte Funktionen problemlos über eigene m-Dateien bzw. Python-Skripte hinzufügen. (Wie unten in Abschnitt 3.5.2 am Beispiel von Boole'schen Ausdrücken gezeigt wird, wäre es im Prinzip auch möglich, einen komplett eigener Parser zu entwickeln und für die Auswertung von Formeln einzusetzen. Dies ist aber für arithmetische Ausdrücke kaum notwendig, da die üblichen Programmiersprachen bereits alle notwendigen Operationen unterstützen.)

Auswertung an diskreten Punkten

Die Überprüfung eines eingebenen Ausdrucks erfolgt dann durch Auswertung dieses Ausdrucks für bestimmte Werte der freien Variablen und Paramenter und Vergleich mit den entsprechenden Werten einer Musterlösung. Die Funktion wird also nur an einigen, diskreten Werten abgetastet, während die im letzten Abschnitt beschriebene symbolische Auswertung eine vollständige Information über die Funktion liefert (bzw. liefern kann). Darüber hinaus treten bei der üblichen numerischen Auswertung mit Gleitkommaarithmetik in jedem Fall Rundungsfehler auf, die geeignet berücksichtigt werden müssen, etwa durch Vorgabe von absoluten oder relativen Toleranzen.

Natürlich kann die Musterlösung ebenso wie die eingegebene Funktion als Ausdruck vorliegen und parallel mit dieser ausgewertet werden. Alternativ ist es aber auch möglich, die Musterlösung nur als vorberechnete Liste der Funktionswerte zu repräsentieren. Dies hat im Kontext des interaktiven Skripts mit seinem freien Zugang zu allen Quelltexten der Vorteil, dass die Studierenden die erwartete Funktion nicht direkt ablesen können. Der in Abbildung 12 gezeigte Umweg über einen Proxy-Server wird daher nur selten notwendig sein.

Entsprechend der Auswahl der auszuwertenden Argumentwerte ergeben sich eine Reihe von Varianten, die sich im Rechenaufwand und der Qualität der Über-

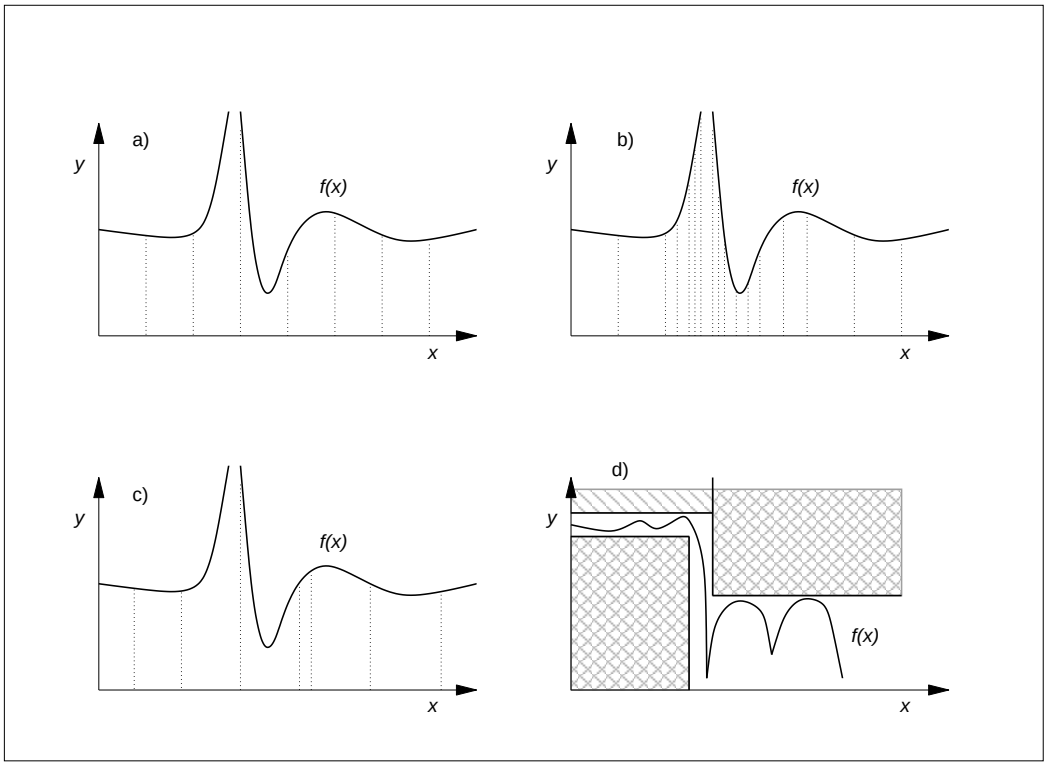


Abbildung 13: *Vier Varianten zur numerischen Überprüfung einer Funktion: a) Abtastung an regelmäßigen Stützstellen, b) adaptiv ausgewählte Stützstellen, c) zufällig ausgewählte Stützstellen, d) Vorgabe von zulässigen bzw. verbotenen Regionen.*

prüfung unterscheiden. Am einfachsten zu realisieren ist die Auswertung auf Stützstellen x_i, die ein vorher geeignet festgelegtes Intervall regelmäßig unterteilen (Abbildung 13a). Der wesentliche Nachteil dieses Ansatzes ist, dass besonders „interessante“ Bereiche einer Funktion nur durch vergleichsweise wenige oder eventuell gar keine Werte abgetastet werden. Dieses Problem kann natürlich durch eine entsprechende Auswahl behoben werden, die mehr Stützstellen für die wichtigen Bereiche der Funktion vorsieht (Abbildung 13b). Diese Auswahl kann statisch vorberechnet werden oder ausgehend von der (numerisch berechneten) Ableitung der Funktion adaptiert werden. Schließlich zeigt Abbildung 13c eine zufällige Auswahl der Stützstellen (Monte-Carlo Varfahren). Dieser Ansatz kommt insbesondere bei mehrdimensionalen Funktionen in Frage, bei denen eine regelmäßige Anordnung der Stützstellen zu sehr grober Abtastung führen würde.

Neben dem punktweisen Vergleich mit einer Musterlösung sind auch weitere Varianten denkbar, insbesondere die Vorgabe von unteren und oder oberen Schranken für ganze Wertebereiche. Dieses Vorgehen wird in Abbildung 13d anhand der Kennlinie eines Tiefpassfilters skizziert.

3.5 Boole'sche Ausdrücke

Wegen ihrer Bedeutung für das Verständnis und den Entwurf digitaler Schaltungen bildet die Boole'sche Algebra einen besonderen Schwerpunkt in der technischen Informatik. Dies gilt auch für die Übungsaufgaben — fast die Hälfte aller Aufgaben zur Vorlesung Technische Informatik 1 behandelt die Umrechnung und Minimierung von Boole'schen Ausdrücken bzw. Funktionen [4].

Leider enthält die Symbolic Math Toolbox von Matlab keine Funktionen zur Manipulation von logischen Ausdrücken, obwohl das zugrundeliegende Programmsystem (Maple) die elementaren Grundfunktionen durchaus bereitstellt. Bei näherer Betrachtung zeigt sich jedoch, dass auch Maple nicht alle der für unsere Übungsaufgaben benötigten Funktionen umfasst. Insbesondere fehlen Funktionen für technisch wichtige Aufgabenstellungen, etwa die Beschränkung auf bestimmte Operationen (z.B. NAND- und NOR-Logik) oder die Minimierung des Realisierungsaufwands. Mangels geeigneter Software muss im Rahmen dieses Projekts daher auf die direkte und automatische symbolische Manipulation von logischen Ausdrücken verzichtet werden.

3.5.1 Überprüfung durch Umwandlung in Normalform

keine Rundungsfehler

Anders als bei der numerischen Verarbeitung von kontinuierlichen Funktionen mit Gleitkommarechnung kann es bei logischen Ausdrücken nicht zu Rundungsfehlern kommen. Eine Überprüfung der Eingaben auf Korrektheit ist daher exakt möglich und erfordert prinzipiell nur den Vergleich mit einer vorgegebenen Musterlösung. Dabei ist natürlich zu berücksichtigen, dass es durchaus auch mehrere gleichwertige Lösungen geben kann.

Normalformen

Der Vergleich mit einer Musterlösung wird dadurch erleichtert, dass eine Reihe von bekannten *Normalformen* existieren, die eine eindeutige Darstellung jeder Boole'schen Funktion erlauben. Zur Überprüfung einer Funktion bzw. eines eingegebenen Ausdrucks reicht es daher aus, die Funktion in die gewünschte Normalform umzuwandeln und dann mit der vorhandenen Musterlösung zu vergleichen. Die folgende Liste zählt einige der verbreiteten Normalformen auf:

- Funktionstabellen
- Disjunktive Normalform (AND-OR)
- Konjunktive Normalform (OR-AND)
- Reed-Muller Form (XOR-AND)
- Reduced-Ordered Binary-Decision Diagrams (ROBDD)

Komplexitätsabschätzungen

Der wesentliche Nachteil bei Verwendung von Normalformen ist die oft exponentielle Komplexität und der damit verbundene Rechenaufwand auch für „einfache" Funktionen. Zwar zeigt eine bekannte Abschätzung von Shannon [43], dass fast alle Boole'schen Funktionen mit n Variablen eine exponentielle Komplexität $O(2^n)$ haben, aber viele in der Praxis wichtige Funktionen wie etwa Addierer lassen sich als Ausdrücke mit lediglich $O(n)$ oder $O(n \ln n)$ Termen schreiben.

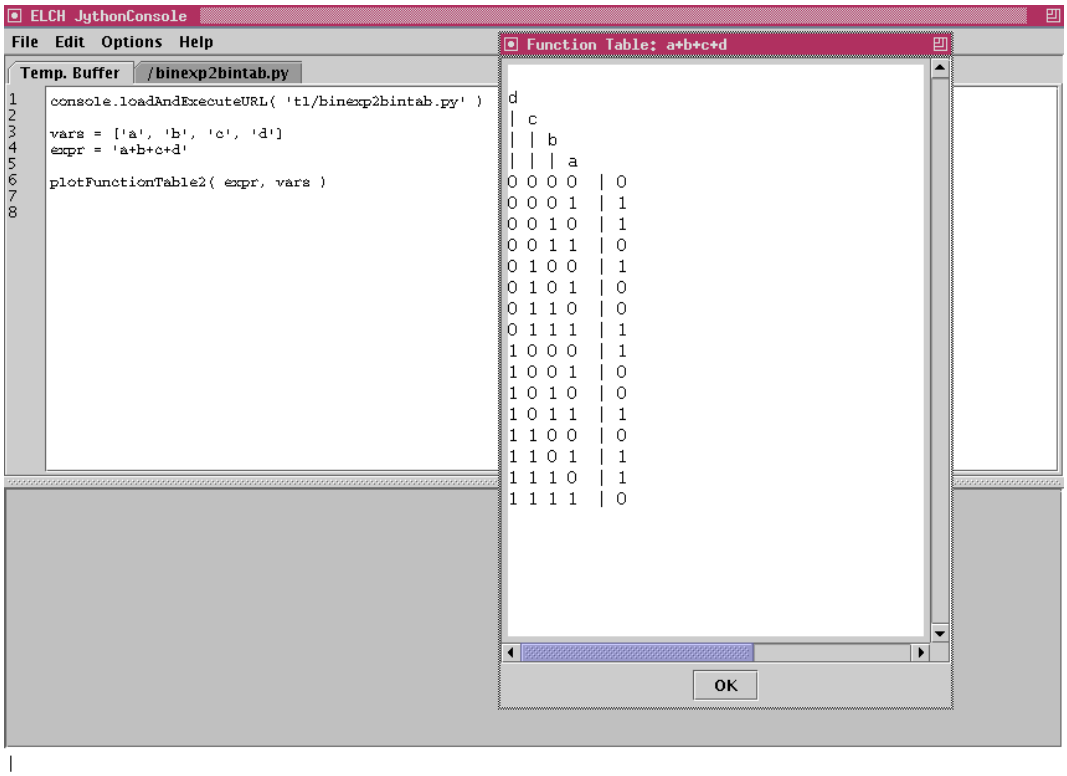


Abbildung 14: *Berechnung einer Funktionstabelle in unseren Jython-Applets. Entsprechende Funktionen stehen auch für die Matlab-Skripte zur Verfügung.*

Für eine umfassende Einführung in die Theorie und Komplexitätsabschätzungen Boole'scher Funktionen sei auf [44] verwiesen. Aber bereits das folgende kleine Beispiel (aus Übungsaufgabe T1.3.6) zeigt die unterschiedliche Komplexität derselben Funktion abhängig von der Darstellung:

$$\begin{aligned} f(x,y,z) &= (\bar{x}\vee y)(\bar{y}\vee z) \\ &= (\overline{xyz})\vee(\overline{xyz})\vee(\bar{x}yz)\vee(xyz) && \text{(DNF)} \\ &= (\bar{x}\vee y\vee\bar{z})(\bar{x}\vee y\vee z)(\bar{x}\vee\bar{y}\vee z)(x\vee\bar{y}\vee z) && \text{(KNF)} \\ &= 1+x+y+xy+yz && \text{(RMF)} \end{aligned}$$

Offenbar umfasst die Funktionstabelle einer Boole'schen Funktion mit n Eingängen immer 2^n Einträge, so dass sich diese Darstellung nur für verhältnismäßig kleine Funktionen (etwa $n \leq 16$ bzw. $2^{16} = 65536$ Termen) eignet. Aber auch die disjunktive und konjunktive Normalform sowie die seltener verwendete Reed-Muller Form erweisen sich in der Praxis als ungeeignet für großes n, da für viele Funktionen ebenfalls eine exponentielle Anzahl der Terme auftritt.

ROBDDs

Dagegen weisen die Reduced-Ordered Binary-Decision Diagrams [1] für viele praxisrelevante Funktionen ein gutartiges Verhalten auf und bilden deshalb auch die Grundlage für aktuelle Programme im professionellen Schaltungsentwurf. Leider sind die meisten dieser Programmpakete nur als C/C++ Versionen verfügbar, während keine geeignete ROBDD-Toolbox für Matlab existiert. Immerhin existieren seit kurzem einige Java-Packages, die bei Bedarf leicht nachträglich in unsere Softwareplattform integriert werden könnten [45].

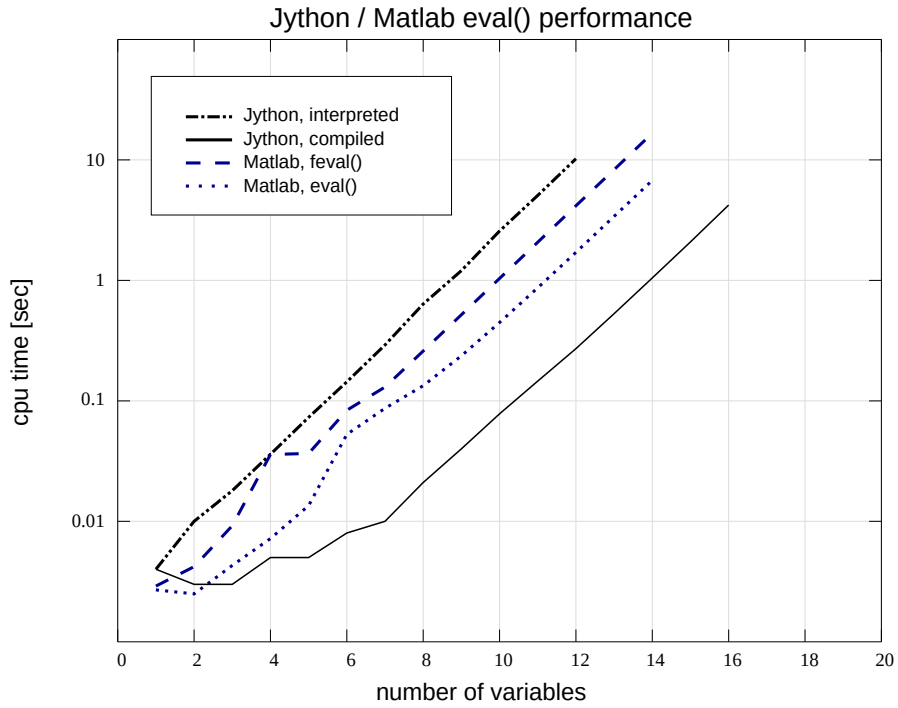


Abbildung 15: *Performance der dynamischen Auswertung von Boole'schen Ausdrücken in Matlab und Jython. Angegeben ist jeweils die Laufzeit zur Berechnung der vollständigen Funktionstabelle einer Testfunktion abhängig von der Anzahl der Variablen. In Matlab wurden die Funktionen eval() und feval() verwendet, in Jython wurde der Ausdruck entweder direkt via eval() ausgewertet (interpretiert) bzw. zur Laufzeit compiliert. Im Beispiel können trotz der exponentiellen Komplexität Funktionen mit bis zu 14 Variablen in weniger als einer Sekunde ausgewertet werden.*

Im Kontext der Überprüfung von Übungsaufgaben spielen solche Komplexitätsbetrachtungen zum Glück keine zentrale Rolle, da praktisch alle in den Aufgaben vorkommenden Funktionen nur wenige ($n < 6$) Variablen umfassen. Tatsächlich basieren die bisher von uns implementierten Algorithmen zur Überprüfung von Boole'schen Funktionen direkt auf Funktionstabellen. Dies erlaubt nicht nur den direkten Vergleich mehrerer Funktionen, sondern ermöglicht bei Abweichungen auch gezielte Hilfestellungen durch einfaches Auflisten der zugehörigen Eingangsbelegungen. Abbildung 14 zeigt exemplarisch die Darstellung einer Funktionstabelle in unseren Jython-Applets.

Performance

Der gegenüber anderen Darstellungen höhere Rechenaufwand zum Erstellen der vollständigen Funktionstabelle muss dafür in Kauf genommen werden. Angesichts der kleinen Problemgröße führt der exponentielle Rechenaufwand selbst bei Verwendung von Skriptsprachen (wie Matlab oder Jython) nicht zu ernsten Performanceproblemen. Abbildung 15 zeigt die exponentielle Abhängigkeit der benötigten Rechenzeit für die Auswertung einer einfachen Beispielfunktion `(a&b&c&d)|(e&f&g&h)|(m&l)` von der Anzahl der aktiven Variablen. Dabei wurden in Matlab die Funktionen `eval()` und `feval()` getestet, während für Jython die direkte Auswertung per Interpreter und die zur Laufzeit compilierte Auswertung eingesetzt wurden. Der compilierte Jython-Code ist am schnellsten und benötigt auf dem Testrechner (2.4 GHz PC) ca. 1 Sekunde Rechenzeit für die Aufstellung der

Funktionstabelle der Testfunktion mit $n = 14$ Eingangsvariablen. Dies ist für die Überprüfung von Übungsaufgaben in jedem Fall ausreichend.

3.5.2 Nebenbedingungen für Boole'sche Ausdrücke

Sowohl der Matlab-Workspace als auch der Jython-Interpreter erlauben die interaktive Eingabe von Ausdrücken und Funktionen als Zeichenketten. In beiden Fällen steht also ein leistungsfähiger Parser zur Verfügung, der die vom Anwender eingegebenen Zeichenketten zur Laufzeit in ausführbaren Programmcode umwandelt bzw. fehlerhafte Eingaben mit detaillierten Fehlermeldungen zurückweist. Leider sind diese Parser in beiden Plattformen in vorgegebenen Funktionen (z.B. `eval()` s.o.) gekapselt und erlauben keinen Zugriff auf die vom Parser erzeugten internen Datenstrukturen. Daher ist es unmöglich, diese internen Datenstrukturen zur weiteren Analyse und Überprüfung von Nebenbedingungen auszunutzen, die für viele unserer Übungsaufgaben gefordert sind. Die folgende Liste nennt einige Beispiele für solche Zusatzanforderungen an die Lösungen:

- Überprüfung der Form des übergebenen Ausdrucks: z.B. Forderung nach diskunktiver, konjunktiver oder Reed-Muller-Form.
- Forderung nach funktionaler anstelle der Infix-Schreibweise, z.B. $XOR(a,b)$ statt $a \oplus b$.
- Überprüfung der Realisierungskosten bei vorgegebener Kostenfunktion, z.B. Anzahl der Logikgatter.
- Einschränkung der Realisierung auf bestimmte Operatoren bzw. Gatter, z.B. `NAND(NAND(1,a),NAND(1,b))` statt $a \vee b$.
- Forderung nach (struktur-) hazardfreier Realisierung.
- Kombinationen dieser Anforderungen.

Zwei Strategien:

Da die meisten dieser Anforderungen nur im Kontext der technischen Informatik bzw. der konkreten technologischen Umsetzung von Boole'schen Funktionen auftreten, bietet keine der üblichen Programmiersprachen eine Unterstützung zur Überprüfung dieser Eigenschaften an. Es war daher notwendig, im Rahmen dieser Projekts einen eigenen Parser für Boole'sche Ausdrücke zu erstellen, der die Analyse der oben aufgezählten Eigenschaften erlaubt.

1. Stringmanipulation

Der einfachste Ansatz dazu, der auch in diversen unserer Matlab-Funktionen und Java-Klassen umgesetzt wurde, besteht in trivialen String-Manipulationen der eingegeben Ausdrücke. Die Grundidee besteht darin, einen eingegebenen Ausdruck im ersten Schritt zunächst vom Parser der jeweiligen Plattform (z.B. Matlab) auswerten zu lassen. Eventuelle Fehler im Ausdruck werden an dieser Stelle erkannt, so dass anschließend keine weitere Syntaxüberprüfung notwendig ist. In den folgenden Schritten können dann die geforderten Eigenschaften durch einfache Stringoperationen überprüft werden. Zum Beispiel kann die disjunktive oder konjunktive Form eines Ausdrucks durch Auftrennung an den Klammern und einfache Suche nach den innerhalb und außerhalb der Klammern vorkommenden Operatorsymbolen erfolgen. Entsprechend ist eine Berechnung der Realisierungskosten durch einfaches Auszählen der Operatorsymbole möglich.

2. Eigener Parser Als Alternative wurde ein komplett eigener Parser für Boole'sche Ausdrücke mit Hilfe des Java-basierten Parsergenerators JavaCC realisiert. In seiner jetzigen Form akzeptiert dieser Parser neben den Operatoren `~`, `&`, `|`, `+` (für Negation, UND, ODER, XOR) auch die entsprechenden Funktionen wie `NOT()`, `AND()`, `NAND()`. Zu jeder dieser Operationen existiert eine zugehörige Java-Klasse, die die jeweilige Operation umsetzt. Der während des Parsens für den eingegebenen Ausdruck erzeugte Baum aus den zugehörigen Java-Objekten kann anschließend bequem analysiert und beliebig modifiziert werden. Aus dieser Implementierung ergeben sich drei Vorteile. Erstens gelingt die Überprüfung der meisten Nebenbedingungen (Realisierungskosten, Art und Anzahl der Operatoren, maximale Tiefe des Baums, usw.) mit trivialen rekursiven Funktionen. Zweitens realisiert der Parser den Zugriff auf die im Ausdruck enthaltenen Variablen über Arrayzugriffe, während Matlab und Jython ihren Namensraum mit Hashtabellen verwalten müssen. Dadurch ergibt sich ein erheblicher Performancevorteil bei mehrfacher Auswertung desselben Ausdrucks für viele Kombinationen der Variablenwerte, was etwa bei Aufstellung der Funktionstabelle erforderlich ist. Drittens umfasst dieser Parser nur ca. 50 KByte an Programmcode und ist auch standalone ohne eine interaktive Umgebung lauffähig. Dies erlaubt es uns, Applets zur Überprüfung von Boole'schen Ausdrücken anzubieten, die wegen der geringen Codegröße sehr schnell geladen werden und damit selbst bei langsamer Internetanbindung (Modem, ISDN) problemlos eingesetzt werden können.

Vorteile

3.5.3 KV-Diagramme

Ein besonders anschauliches Hilfsmittel zur Manipulation und zur Minimierung von Schaltfunktionen basiert auf den sogenannten Karnaugh-Veitch-Diagrammen [4]. Wegen der geschickten Anordnung der einzelnen Terme in diesen Diagrammen sind mögliche Vereinfachungen durch Gruppierung von Termen unmittelbar „auf einen Blick“ zu erkennen, so dass sich das Verfahren ideal als Einführung in die Thematik der Minimierung des Realisierungsaufwands logischer Schaltungen eignet.

kvdiagram.m Mit der Funktion `kvdiagram` steht ein interaktives Applet zur Eingabe und Minimierung von KV-Diagrammen direkt als Komponente des Matlab-Skripts zur Vorlesung Technische Informatik 1 zur Verfügung. Es unterstützt die Erzeugung und Visualisierung von Diagrammen mit beliebig vielen Variablen und erlaubt die direkte Manipulation sowohl in disjunktiver als auch konjunktiver Form. In der Praxis eignen sich aber nur Funktionen mit bis zu sechs Variablen für die KV-Diagramme, da für komplexere Funktionen die Übersichtlichkeit schnell wieder verlorengeht und eine zweistufige Realisierung kaum in Betracht kommt. Zusammen mit der Hilfsfunktion `checkkvmin`, die die Anzahl der für ein KV-Diagramm benötigten Gatter berechnet und mit dem vorgegebenen Realisierungsaufwand für eine Musterlösung vergleicht, ist die Überprüfung aller Übungsaufgaben zu KV-Diagrammen direkt im interaktiven Skript möglich.

Als Alternative wurde im Rahmen des Projekts zusätzlich ein früher im Rahmen einer Studienarbeit erstelltes Java-Applet [13] aktualisiert und an die Konventionen der Vorlesung T1 angepasst. Dies betrifft die Umstellung auf die Swing-GUI-Bibliotheken und eine andere Berechnung des Realisierungsaufwands (Anzahl der Gatter anstelle der Anzahl der Gattereingänge). Dieses Applet kann problemlos in HTML-Webseiten eingebettet werden und ist auch ohne Matlab-Lizenz lauffähig.

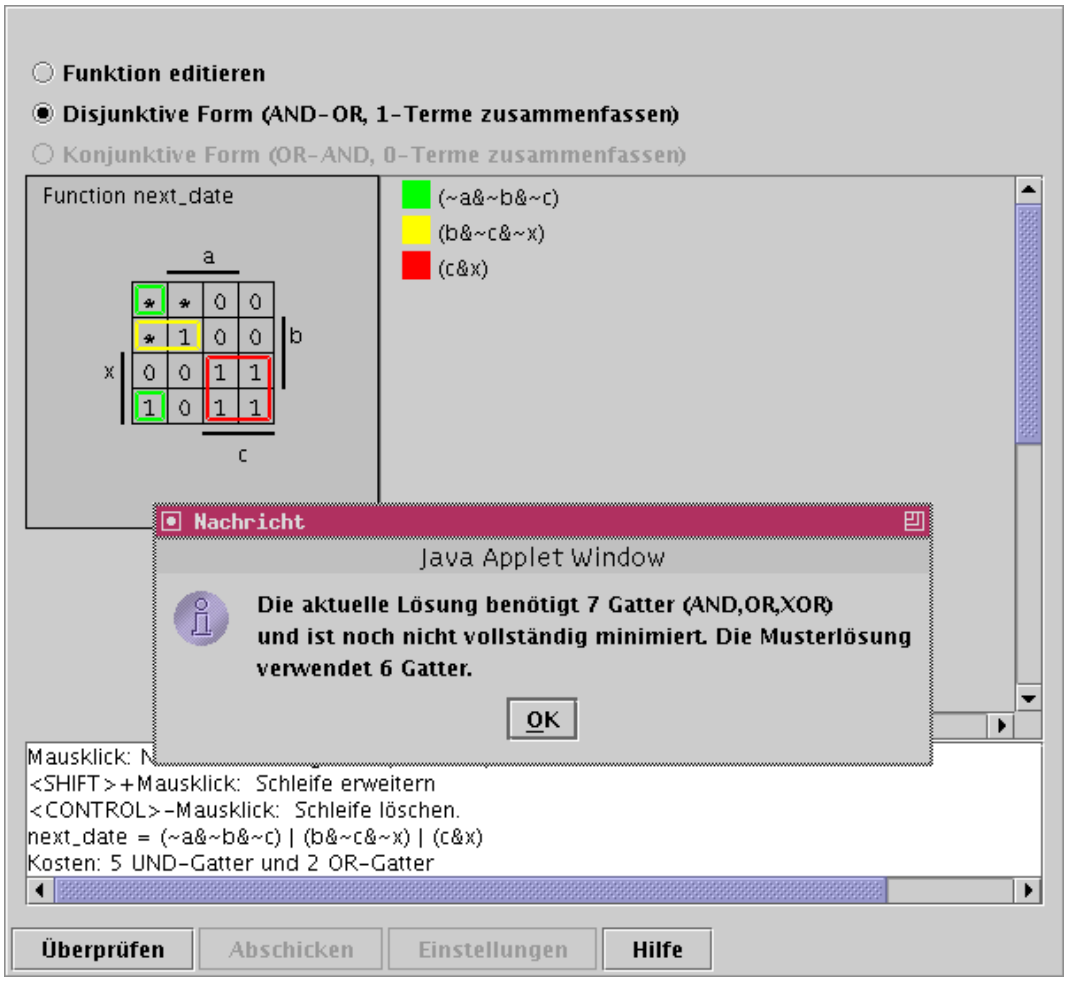


Abbildung 16: *Interaktive Manipulation und Minimierung einer Schaltfunktion mit KV-Diagrammen. Das Applet vergleicht von den Studierenden interaktiv eingegebene und in disjunktiver oder konjunktiver Form minimierte Schaltfunktion mit einer vorgegebenen Musterlösung.*

Abbildung 16 sowie die Abbildung auf der Titelseite zeigen Screenshots dieses Applets bei Bearbeitung von Funktionen mit vier bzw. mit sechs Eingangsvariablen. Wie bei der üblichen Darstellung mit Papier und Bleistift werden die zusammengefassten Terme dabei durch farbig markierte Schleifen dargestellt.

Als Hilfestellungen liefern beide Version der KV-Diagramm-Applets Hinweise auf Abweichungen von der Musterlösung, die als Funktionstabelle mit optionalen Don't-Care Termen vorgegeben wird. Die Überprüfung des Realisierungsaufwands erfolgt jeweils durch Abzählen der benötigten Gatter und Vergleich mit der vorgegebenen Gatteranzahl, vgl. Abbildung 16. Dieser Algorithmus akzeptiert auch die möglichen Lösungsvarianten, die durch andere Zusammenfassung von Termen entstehen können und zu gleichwertigen Lösungen führen.

3.5.4 Ein graphischer Editor für zweistufige Schaltungen

Schaltbilder

Neben der textuellen Darstellung von Boole'schen Ausdrücken und der Visualisierung mit KV-Diagrammen wird drittens auch die graphische Repräsentation mit Schaltbildern häufig verwendet. Ein Vorteil der graphischen Darstellung ist, dass

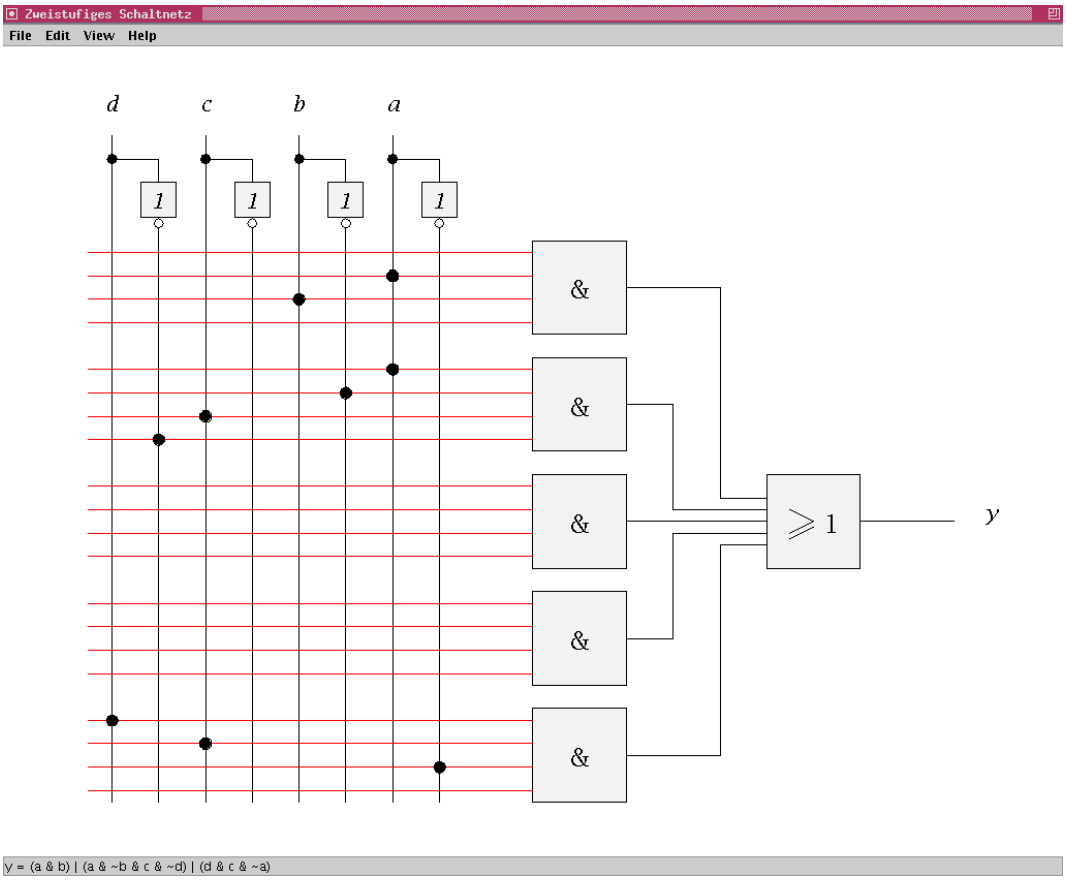


Abbildung 17: *Der in der Abbildung gezeigte Editor für zweistufige Schaltungen erlaubt die interaktive Eingabe einer Schaltung durch Anklicken der Kreuzungspunkte zwischen den Eingangsleitungen (vertikal) und den Eingängen der Gatter (horizontal). Aus den vom Anwender gesetzten Verbindungen wird der zugehörige Boole'sche Ausdruck berechnet und steht dann zur Überprüfung zur Verfügung.*

viele in der Praxis relevante Funktionen quasi auf einen Blick erkannt werden können. Voraussetzung für die automatische Überprüfung und Hilfestellung ist offenbar ein geeigneter Schaltplaneditor. Bereits im letzten Projektbericht [12] wurde erläutert, wie die Überprüfung durch Simulation und Signaturanalyse realisiert werden kann.

Spezialfall zweistufige Schaltungen

Für einfache Schaltungen ist dieses Vorgehen jedoch unnötig kompliziert. Als Alternative wurde ein zusätzlicher kleiner Graphikeditor speziell für die Eingabe von *zweistufigen Schaltungen* geschrieben, bei dem die Grundstruktur der Schaltung mit (derzeit) vier Eingangsvariablen und fünf Gattern in der ersten Stufe fest vorgegeben ist. Dieser in Abbildung 17 gezeigte Editor erlaubt das interaktive „Zusammenklicken" einer Schaltung durch einfaches Anklicken der Kreuzungspunkte zwischen den vertikal verlaufenden Eingangsleitungen und den horizontalen Verbindungen zu den Gattern der ersten Stufe. Alternativ zur in der Abbildung gezeigten disjunktiven Form (AND-OR) ist wiederum auch die konjunktive Form (OR-AND) möglich.

viele gleichwertige Lösungen

Es ist zu beachten, dass trotz der festgelegten zweistufigen Grundstruktur der Schaltung immer noch viele gleichwertige Lösungen für jede zu realisierende logische Funktion existieren, die sich durch Permutation der Eingangsleitungen und Gatter ineinander überführen lassen. Dies bedeutet natürlich, dass ein trivialer 1:1-

Vergleich mit einer einzelnen Musterlösung nicht zur Überprüfung in Frage kommt. Statt dessen wird vom Editor nach jeder Änderung einer Verbindung automatisch der zugehörige logische Ausdruck neu berechnet und steht als String zur Verfügung; in der Abbildung ergibt sich der Ausdruck $y = (a \wedge b) \vee (a \wedge \neg b \wedge c \wedge \neg d) \vee (\neg a \wedge c \wedge d)$. Dies erlaubt es insbesondere, die in Abschnitt 3.5 vorgestellten Algorithmen zur Überprüfung zu verwenden.

Die gewählte Darstellung der Schaltpläne entspricht übrigens der in Datenblättern üblichen Darstellung von programmierbaren Logikbausteinen wie GALs [41] und dient damit gleichzeitig als Vorbereitung auf den Umgang mit diesen Bausteinen.

4 Überprüfung von Programmen

Nicht zuletzt die täglich neuen Meldungen über Sicherheitslücken in Anwendungsprogrammen und Betriebssystemen zeigen die Bedeutung der Überprüfung der Korrektheit und Sicherheit von Programmen. Trotz jahrelanger Forschungsarbeiten auf diesem Gebiet beschränken sich die verfügbaren Ansätze aber immer noch auf kleine Teilbereiche, etwa bei Verwendung bestimmter Programmiersprachen oder Werkzeuge [36, 37]. Als Beispiel sei auf das ebenfalls via ELCH geförderte Projekt INCOM [38] hingewiesen, das sich speziell auf die Analyse von Programmen in deklarativen Sprachen beschränkt.

Abstraktionsebenen

Wie bereits bei Vorstellung unserer Klassifikation von Übungsaufgaben erläutert wurde [11], lässt sich das gesamte Themengebeit der Softwareentwicklung zwanglos durch die behandelten Abstraktionsebenen untergliedern (vgl. Abbildung 1 auf Seite 3). Im Rahmen der Vorlesungen zur technischen Informatik kommen dabei nur die untersten Abstraktionsebenen vor. Dies sind die direkte Programmierung von Hardware mittels Microcode, die Programmierung mit Maschinensprache und schließlich eine Einführung in die Assemblerprogrammierung. Die darüberliegenden Ebenen dagegen sind den Einführungsvorlesungen zur Programmierung bzw. praktischen Informatik sowie deren Übungen vorbehalten.

Übersicht

Der folgende Abschnitt 4.1 beschreibt zunächst den Aufbau unseres Demonstrationsrechners „PRIMA“ und den im Rahmen des Projekt entwickelten Simulator für diese Maschine. In Abschnitt 4.2 wird erläutert, wie die von den Studierenden entwickelten Programme für diese Maschine durch Simulation überprüft werden können. Schließlich beschreibt Abschnitt 4.3 eine Reihe von Ansätzen zur Hilfestellung während der Programmentwicklung.

4.1 Die Primitive Maschine PRIMA

bottom-up

In den meisten Lehrbüchern und Vorlesungsskripten zur Thematik der digitalen Systeme wird das Thema Rechnerarchitektur und speicherprogrammierbarer Rechner („von-Neumann Rechner“) mit einem *bottom-up*-Ansatz eingeführt. Aufbauend auf der Boole’schen Algebra werden zunächst die Grundkomponenten eines Rechners (Multiplexer, Rechenwerke, Register, etc.) vorgestellt. Dann wird aus diesen Grundkomponenten ein einfacher Mikrorechner aufgebaut und dessen Funktionsweise anhand der Ausführung einzelner Befehle erläutert. Anschließend werden einfache Maschinen-Programme für diesen Rechner vorgeführt und die Grundprinzipien der Assembler-Programmierung erläutert.

Die PRIMA

Dieses bewährte Vorgehen wird auch in unserer Vorlesung eingesetzt [4, 5]. Als Grundlage dient die *Primitive Maschine* (PRIMA), eine sehr einfache Akkumulatormaschine mit lediglich vier Registern und 256 Worten Hauptspeicher bei 8-bit Wortbreite. Der Befehlssatz umfasst ausschließlich die grundlegenden Rechenoperationen, Lade- und Speicherbefehle mit direkter Adressierung sowie einige Sprungbefehle. Durch die horizontale Befehlskodierung reichen wenige Gatter zur Realisierung des Rechners aus, die gesamte Schaltung lässt sich bequem auf einem Blatt Papier skizzieren. Der einfache Befehlszyklus der Maschine umfasst jeweils drei Takte (Befehl holen, Adresse holen, Befehl ausführen). Lediglich das Inkrementieren des Befehlszählers erfolgt implizit parallel zu den übrigen Operationen, was die

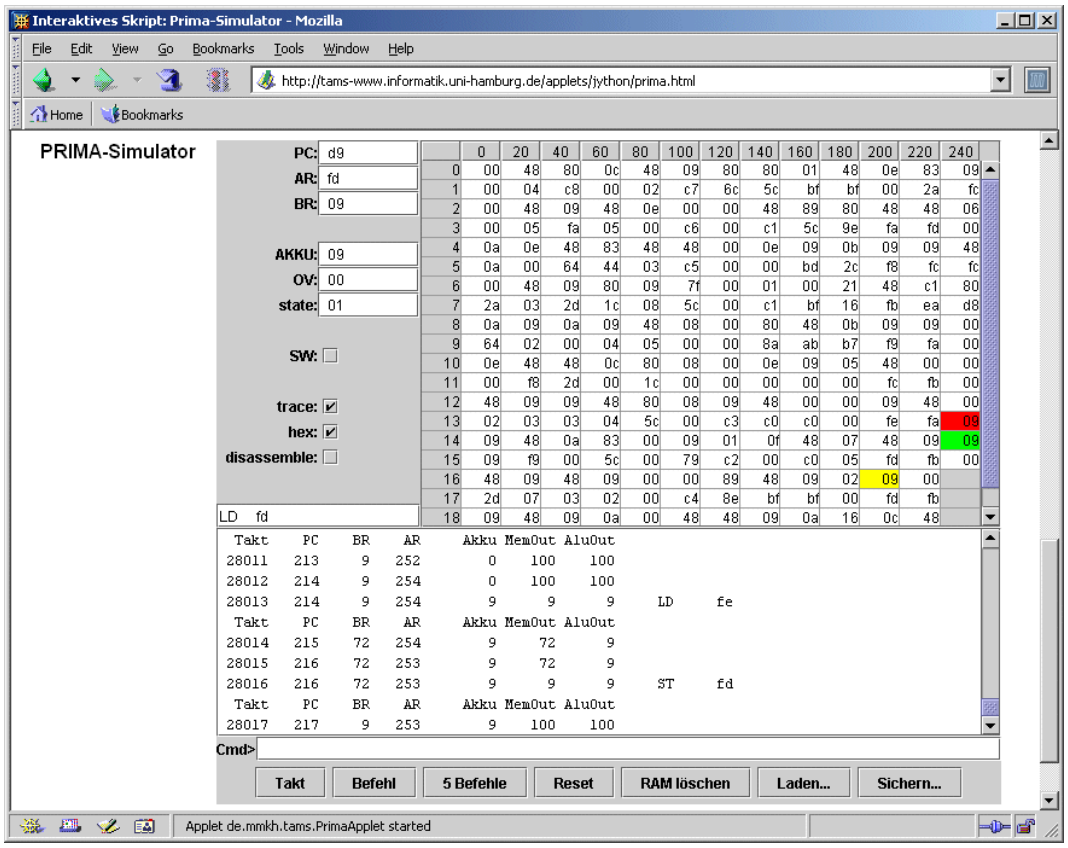


Abbildung 18: *Simulator für die primitive Maschine (PRIMA) als Java-Applet. Neben Register- und Speicherinhalten wird auf Wunsch ein Ablaufprotokoll dargestellt; auch eine Disassemblierung der Speicherinhalte ist möglich.*

Beobachtung der Zeitabläufe stark erleichtert. Da komplexere Adressierungsarten fehlen, müssen viele Operationen wie z.B. Arrayzugriffe mittels selbstmodifizierendem Code realisiert werden.

Der besonders einfache Aufbau der PRIMA erleichtert nicht nur das Verständnis für die Studierenden, sondern auch die Simulation. Derzeit verfügen wir über mehrere Simulatoren der PRIMA, sowohl als einfache standalone-Applikationen auf der Befehlsebene bis hinunter zu einer Simulation auf Gatterebene. Leider konnte ein bereits 1996 realisiertes, stark graphisch orientieres Applet [40] didaktisch nicht überzeugen. Die in diesem Applet eingesetzten Animationen zur Darstellung der einzelnen Datentransfers sehen zwar hübsch aus, lenken aber eher vom Verständnis der Vorgänge ab als dieses zu unterstützen.

Simulator

Abbildung 18 zeigt den in Rahmen dieses Projekt neu entwickelten Simulator, der als kleines Java-Applet realisiert ist und ohne weitere Installation benutzt werden kann. Die graphische Oberfläche stellt die Register- und Speicherinhalte des Rechners dar; diese können alternativ als dezimale oder hexadezimale Werte dargestellt werden und lassen sich direkt editieren. Neben der Ausführung einzelner oder meherer Befehle lassen sich auch die einzelnen Taktschritte simulieren, wobei die zugehörigen Operationen auf Wunsch im mittleren Textfeld protokolliert werden, um die Zeitabläufe genau mitzuverfolgen. Als weitere Hilfestellung werden die zuletzt zugegriffenen Speicherstellen farblich hervorgehoben, so dass Lese- und Schreibzugriffe während der Simulation unmittelbar deutlich werden.

und Assembler

Zusammen mit dem neuen Simulator wurde auch ein einfacher Assembler erstellt, der die Programmentwicklung für die PRIMA gegenüber der reinen Maschinensprache erheblich erleichtert. Da die Studierenden aber auch die Programmierung auf der reinen Maschinensprache kennenlernen sollen, ist noch entschieden, ob und wann dieser den Studierenden zur Verfügung gestellt wird.

4.2 Überprüfung von Maschinenprogrammen

Syntaktische oder semantische Analysen ...

Anders als bei Verwendung von höheren Programmiersprachen mit ihren ausgefeilten Syntaxregeln und Typsystemen ergeben sich auf der Ebene der Maschinen- und Assemblerprogramme kaum Ansatzpunkte für eine aufwendige Analyse der Programme. Da Maschinenprogramme letztlich nur ein einfaches Speicherabbild des jeweiligen Rechners sind, ergibt sich vielmehr die Situation, dass praktisch alle der überhaupt möglichen Kombinationen (bei der PRIMA immerhin 2^{2048}) gleichzeitig legale Programme sind. Eine rein syntaktische Überprüfung ist damit sinnlos. Auf der anderen Seite ist eine semantische Analyse praktisch nicht durchführbar, zumal wegen des eingeschränkten Befehlssatzes der PRIMA fast immer selbstmodifizierende Programme nötig sind.

Derzeit existiert nur eine einzige Ausnahme, wo aufwendige Algorithmen zur semantischen Analyse von Maschinencode eingesetzt werden. Dies ist der sogenannten *Bytecode Verifier* als Komponente der Java Virtual Machine, der dazu dient, unbekannten und potentiell bösartigen Programmcode vor der Ausführung auf Konsistenz und Typsicherheit zu überprüfen. Eine ausführliche Diskussion der für diesen Verifier eingesetzten Techniken findet sich u.a. in [42]. Allerdings verfügt die die Java Virtual Machine über einen speziell zum Zweck der Überprüfbarkeit entworfenen Befehlssatz und setzt eine aufwendige Speicherverwaltung mit diversen sauber getrennten Speicherbereichen voraus. Leider lassen sich die für die Java VM eingesetzten Techniken deshalb nicht auf beliebige PRIMA-Programme übertragen und anwenden.

Auch auf der Ebene des Assemblerprogrammierung stellen Syntaxfehler erfahrungsgemäß keine Hürde für die Studierenden dar, da die einzigen Fehlermöglichkeiten wie undefinierte Opcodes oder Symbole bereits vom Assembler durch geeignete Fehlermeldungen zurückgewiesen werden.

und Simulation

Deshalb verbleibt wiederum die Simulation als einfachste Lösung zur Überprüfung der Programme. Dazu werden die von den Studierenden entwickelten Programme zusammen mit den notwendigen Eingabedaten in den Simulator geladen. Danach wird das Programm im Simulator ausgeführt, bis über eine geeignete Abbruchbedingung das Programmende erkannt wird oder eine vorher festgelegte Maximalanzahl der Befehle überschritten wurde. Schließlich werden die vom Programm berechneten Werte mit den erwarteten Ergebnissen verglichen. Dies ist möglich, sofern die Aufgabenstellung ausreichend präzise formuliert ist, z.B. durch explizite Vorgabe der Speicheradressen für die Eingaben und Ausgaben des Programms.

Angesichts der sehr einfachen Struktur von typischen Demonstrationsrechnern wie der PRIMA lassen sich auf aktuellen PCs leicht einige Millionen simulierter Befehle pro Sekunde berechnen. Das oben skizzierte Vorgehen bereitet deshalb keine Performanceprobleme. Für eine Reihe von Übungsaufgaben dürfte es sogar möglich

```
1  ; Aufgabe T1.6.2.a:
2  ; PRIMA Maschinenprogramm für die Berechnung der Fakultät n!
3  ; mit 0 <= n <= 5 mit Nachschlagen in einer Tabelle.
4  ;
5  ; Label Mnemonic   Adr.  Wert.  Kommentar
6
7  start:
8         LD  50      0      9   ; Eingabewert n laden
9                     1     50   ;
10        ADD 19      2      0   ; Startadresse der Tabelle addieren
11                    3     19   ;
12        ST   7      4     72   ; Index abspeichern (mitten im
13                    5      7   ; Programm: siehe nächsten Befehl)
14        LD   0      6      9   ; Resultatwert laden (Adresse wurde
15                    7      0   ; vom vorigen Befehl geschrieben!)
16        ST   51     8     72   ; Resultatwert n! abspeichern
17                    9     51   ;
18
19 end:
20        JMP  end   10    128   ; Endlosschleife als Programmende
21                   11     10   ;
22
23 table:
24                   19     20   ; Startadresse der Tabelle
25                   20      1   ; 0!
26                   21      1   ; 1!
27                   22      2   ; 2!
28                   23      6   ; 3!
29                   24     24   ; 4!
30                   25    120   ; 5!
31
32 inout:
33                   50      4   ; Eingabe, hier n=4  (0..5)
34                   51      0   ; Resultat n!
```

Abbildung 19: *Maschinenprogramm für die PRIMA mit Lookup-Tabelle als Lösung für Aufgabe T1.6.2a. Erläuterung im Text.*

sein, die Simulation für alle möglichen Eingabedaten durchzuführen und damit eine vollständige Überprüfung der Programme zu erreichen. Falls die Anzahl der möglichen Eingaben zu groß sein sollte, wird sich die Simulation auf einen Satz relevanter Eingabedaten sowie einige zusätzliche zufällig generierte Eingabedaten beschränken müssen. Dieses Vorgehen lässt sich am besten mit einem Beispiel illustrieren:

> **Aufgabe T2-6.2a**: Es ist ein Prima-Programm für die Berechnung der Fakultät $n!$ (mit $0 \leq n \leq 5$) zu entwicklen: mit Nachschlagen in einer Tabelle.

```
; Jython-Skript zur Überprüfung von Aufgabe T1.6.2a

; PRIMA Simulator erzeugen, zu testendes Programm laden
;
prima = de.mmkh.tams.Prima()
prima.clearRAM()
prima.loadRAM( 't1-6-2a-markus-mustermann.txt' )

; Eingaben setzen, Programm ausführen, Resultate vergleichen
;
args     = [0, 1, 2, 3,  4,   5]
expected = [1, 1, 2, 6, 24, 120]
for i in args:
  prima.setRAM( 50, args[i] )
  prima.reset()
  for t in range(0,100):
    prima.cycle()
  if (prima.getRAM(51) != expected[i]):
    message( 'Falsches Ergebnis für n=' + str(i) )
    return
message( 'Programm funktioniert korrekt' )
```

Abbildung 20: *Skript zur automatischen Überprüfung eines eingeschickten Programms durch Simulation und Wertevergleich mit der Musterlösung. Erläuterung im Text.*

Der Schlüssel zur Lösung dieser Aufgabe liegt in der Umsetzung der notwendigen indizierten Adressierung durch selbstmodifizierenden Code: das Programm muss die korrekte Adresse zum Zugriff auf das benötigte Element der Tabelle berechnen und diese Adresse dann in einem Ladebefehl eintragen. Da die Funktionstabelle insgesamt nur 6 Werte umfasst (0!=1, 1!=1, 2!=2, 3!=6, 4!=24, 5!=120) ist das zugehörige Programm sehr einfach zu erstellen. Eine mögliche Lösung ist in Abbildung 19 dargestellt. Das Programm besteht dabei nur auf sechs Befehlen und einigen Bytes für die geeignet initialisierten Datenbereiche.

Das Grundgerüst eines Skriptes zur automatischen Überprüfung ist in Abbildung 20 skizziert; die vollständige Version verfügt über ausführlichere Meldungen. Einige Erweiterungen zur Überprüfung von aufwendigeren Programmen liegen auf der Hand: insbesondere ist es wünschenswert, die Anzahl der zu simulierenden Zyklen nicht fest zu kodieren, sondern abhängig von den Eingabewerten einzustellen oder das Programmende zu erkennen.

Zusätzlich ist es möglich, strukturelle Merkmale des Programms zu überprüfen, etwa ob bestimmte Befehle wie Multiplikation, Shifts, oder bedingte Sprünge überhaupt vorkommen. Wegen der selbstmodifizierenden Programme ist diese Art der Überprüfung bei der PRIMA allerdings nur eingeschränkt möglich, da die gesuchten Befehle oder Adressen eventuell erst während des Programmablaufs geschrieben werden. Bei anderen Rechnerarchitekturen, zum Beispiel bei RISC-Architekturen mit ihrem sauberen Befehlssatz und der üblichen vollständigen Tren-

nung des Speichers in separate Bereiche für Programmcode und Daten dürften diese Tests dagegen eine willkommene Unterstützung der Überprüfung bieten.

4.3 Hilfestellungen

Auch im Kontext der Rechnerarchitektur ergibt sich die wichtigste Hilfestellung für die Lernenden wieder durch die Verfügbarkeit einer *interaktiven Umgebung*: Im Simulator können die Programme schrittweise entwickelt und sofort getestet werden. Dabei sind viele typischen Fehler unmittelbar zu erkennen und lassen sich leicht korrigieren. Dies gilt insbesondere für Flüchtigkeitsfehler, nicht zuletzt Schreibfehler in Befehlen und Adressen, oder die häufigen off-by-one Fehler bei Schleifentests und Arrayzugriffen.

Wie im letzten Abschnitt erläutert wurde, ist die automatische Analyse von Maschinen- und Assemblerprogrammen praktisch nicht zu leisten. Trotzdem kann die interaktive Umgebung durch automatische Hilfestellungen ergänzt werden, die auf Plausibilitätstests beruhen. Dabei kommen sowohl statische Überprüfungen anhand des Programmtextes als auch dynamische Tests während der Programmausführung in Frage. Die folgende Auszählung nennt einige Möglichkeiten:

- Überprüfung aller Befehle auf gültiger Opcode und korrekte Speicherausrichtung; zum Beispiel verwendet die PRIMA ausschließlich 2-Wort Befehle, die per Konvention auf geraden Adressen liegen sollten.
- Überprüfung der Gültigkeit der Zieladressen bei Sprungbefehlen; bei Assemblerprogrammen zum Beispiel nur auf Adressen, die auch als Sprungmarken definiert sind.
- Vorgabe von bestimmten Adressbereichen für Programmcode und (evtl. mehrere separate) Datenbereiche. Erkennen und Abfangen von Lese- und Schreibzugriffen auf Speicherstellen außerhalb dieser Adressbereiche.
- Erkennen möglicher Probleme durch Überschreiben von Programmcode während der Programmausführung. Auch für selbstmodifizierende Programme kann überprüft werden, ob nur die Adressen oder auch Befehle geschrieben werden.
- Überprüfung, ob alle Eingabedaten gelesen werden und als Ausgabewerte geschrieben werden.
- Test auf Plausibilität aufeinanderfolgender Befehle, wobei diese Abhängigkeiten für die jeweilige Zielarchitektur vorberechnet werden können. Auf diese Weise lassen sich unter anderem sinnlose Befehlsfolgen erkennen, etwa ein bedingter Sprung direkt hinter einem Befehl, der sich konstant auf die jeweilige Sprungbedingung auswirkt (bei der PRIMA etwa LD1, BZ).

5 Zusammenfassung

Das interaktive Skript verfolgt den Ansatz, erläuternden Text nicht nur mit statischen Medien aufzubereiten sondern direkt mit interaktiv ausführbarem Programmcode zu kombinieren. Dabei können nicht nur die Parameter sondern auch die zugrunde liegenden Funktionen selbst vom Anwender modifiziert und ergänzt werden, womit ideale Voraussetzungen für entdeckendes Lernen gegeben sind.

Als Software-Plattform für unsere Vorlesungen zur technischen Informatik dient zunächst Matlab. Das Prinzip der interaktiven Skripte lässt sich aber mit jeder Programmierumgebung umsetzen, die das interaktive Ausführen von Programmtexten oder Skripten erlaubt. Unser Konzept zur Einbettung von kleinen Java-Applets in interaktive HTML-Seiten ist hier eine interessante Alternative, da alle Funktionen der bekannten Web-Browser zur Verfügung stehen und die Anwender in ihrer gewohnten Umgebung arbeiten. Mit dem in Abschnitt 2.3 beschriebenen Konverter konnten bisher drei vollständige Vorlesungen mitsamt aller Materialien nach HTML umgesetzt werden.

Die Integration von Übungsaufgaben in die interaktiven Skripte erlaubt nicht nur eine Kontrolle des Lernfortschritts, sondern ist eine zentrale Säule des didaktischen Konzepts. Inhalt und Ziel des vorliegenden Projekts ist die Realisierung von Verfahren, um die Studierenden beim Bearbeiten der Übungsaufgaben zu unterstützen und gleichzeitig die Übungsgruppenleiter von Routineaufgaben zu entlasten.

In diesem Report wurden Verfahren und Algorithmen für die im ersten Bericht [11] aufgestellte Klasse der Formel-Aufgaben vorgestellt: Neben den Verfahren zur Überprüfung der Lösungen auf Korrektheit wurden jeweils auch Ansätze zu Hilfestellungen vorgestellt, um die Studierenden gezielt auf falsche Lösungsansätze oder Flüchtigkeitsfehler hinzuweisen, ohne jedoch die Lösungen selbst vorwegzunehmen.

Da die im Projekt entwickelten Algorithmen und Verfahren eine Vielzahl von möglichen Übungsaufgaben abdecken und die Software ausreichend flexibel ausgelegt ist, werden sich die Ergebnisse auch zur Unterstützung von Übungen und Praktika in anderen Fachgebieten einsetzen lassen.

Literatur

[1] R. E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation,* IEEE Transactions on Computers, C-35(8):677–691, 1986

[2] R. E. Bryant, D. R. O'Hallaron, *Computer Systems, A Programmer's Perspective,* Prentice Hall, 2003, ISBN 0-13-034074-X

[3] Fachbereich Informatik der Universität Hamburg, *Studienführer Informatik 2002/2003*, www.informatik.uni-hamburg.de/

[4] K. v. d. Heide, *Vorlesung Technische Informatik 1*, Universität Hamburg, FB Informatik, WS2002, tams-www.informatik.uni-hamburg.de/lehre/ws2002/t1Vor/

[5] K. v. d. Heide, *Vorlesung Technische Informatik 2*, Universität Hamburg, FB Informatik, SS2003, tams-www.informatik.uni-hamburg.de/lehre/ss2003/vorlesungen/T2/

[6] K. v. d. Heide, M. Grove, *Übungsaufgaben und Musterlösungen zur Vorlesung Technische Informatik*, Universität Hamburg, FB Informatik, SS2003, tams-www.informatik.uni-hamburg.de/onlineDoc/

[7] K. v. d. Heide, M. Grove, N. Hendrich, B. Wolfinger, *Praktikum Technische Informatik 1-4*, Universität Hamburg, FB Informatik, tams-www.informatik.uni-hamburg.de/onlineDoc/

[8] N. Hendrich, *Hades Tutorial*, tams-www.informatik.uni-hamburg.de/applets/hades/archive/tutorial.pdf

[9] N. Hendrich, *A Java-based framework for simulation and teaching*, Proc. 3rd European Workshop on Microelectronics Education, EWME-2000, 285–288, Aix en Provence, 2000

[10] N. Hendrich, *Automatic checking of students' designs using built-in selftest methods*, Proc. 3rd European Workshop on Microelectronics Education, EWME-2002, Baiona, 2002

[11] N. Hendrich, K. v.d.Heide, *Automatische Überprüfung und Hilfestellung zu Vorlesungs-begleitenden Übungsaufgaben*, Projektbericht, Multimedia-Kontor Hamburg, 2003

[12] N. Hendrich, K. v.d.Heide, *Automatische Überprüfung und Hilfestellung zu Vorlesungs-begleitenden Übungsaufgaben*, Projektbericht, Multimedia-Kontor Hamburg, 2004

[13] M. Mayer, *Konzeption und Umsetzung eines Java-Applets zur Logikminimierung mit KV-Diagrammen*, Studienarbeit, Fachbereich Informatik, 1998

[14] J. L. Hennessy, D. A. Patterson, *Computer organization and design: the hardware/software interface*, Morgan Kaufmann, 1998, ISBN 1-558-60491-X

[15] J. Hugunin, *Python and Java — The Best of Both Worlds*, Proc. 6th Internation Python Conference, San Jose, 1997,

[16] D. Jansen (Hrsg.), *Handbuch der Electronic Design Automation* Hanser, 2001 ISBN 3-446-21288-4

[17] *Jython Environment for Students*, Georgia Institute of Technology, 2002 http://coweb.cc.gatech.edu/cs1315/814

[18] Jython project homepage, www.jython.org

[19] J.W. Eaton and others, *GNU Octave Project*, www.octave.org

[20] The MathWorks, Inc., *Matlab Version 5 User's Guide*, 1997 ISBN 0-13-272550-9

[21] The MathWorks, Inc., *Matlab Version 6 User's Guide*, 2002

[22] W. Schiffmann, R. Schmitz, *Technische Informatik 1*, Springer Verlag, 2001, ISBN 3-540-42170-X

[23] W. Schiffmann, R. Schmitz, *Technische Informatik 2*, Springer Verlag, 1999, ISBN 3-540-

[24] W. Schiffmann, R. Schmitz, *Übungsbuch zur Technischen Informatik 1 und 2* (2. Auflage), Springer Verlag, 2001, ISBN 3-540-42171-8

[25] R. W. Schmidt, *Java Network Launching Protocol & API Specification*, JSR-56, Sun Microsystems, Inc., 2001,

[26] R. Schulmeister, *Grundlagen hypermedialer Lernsysteme: Theorie – Didaktik – Design*, Oldenbourg, 1997, ISBN 3-486-24419-1

[27] A. S. Tanenbaum, *Structured Computer Organization, 4th. Edition*, Prentice Hall, 1999 ISBN 0-13-020435-8

[28] imc information multimedia communication AG, *Clix 4 Campus* Lernplattform, http://www.im-c.de/homepage/clix_campus.htm

[29] World wide web consortium, *Extensible Markup Language (XML)*, http://www.w3.org/XML/

[30] P. Hubwieser, *Didaktik der Informatik*, Springer 2000, ISBN 3-540-43510-7

[31] H. Wojtkowiak, *Test und Testbarkeit digitaler Schaltungen*, Teubner 1988, ISBN 3-519-02263-X

[32] N. Freed, N. Borenstein, *Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies*, Network Working Group, RFC 2045, www.ietf.org/rfc/rfc2045.txt

[33] D. Raggett, A. Le Hors, I. Jacobs, Eds. *World-wide web consortium, HTML 4.01 Specification*, W3C Recommendation 24 December 1999, http://www.w3.org/TR/html401

[34] S. Emmerson, Java Specification Requests, *JSR 108: Units Specification*, http://www.jcp.org/en/jsr/detail?id=108, http://jade.dautelle.com

[35] A. Ruge, *Ein HTML-Browser für interaktive Lehrbücher*, Diplomarbeit, Fachbereich Informatik, Universität Hamburg, 2004

[36] K. Schneider, *Verificatin of reactive systems: formal methods and algorithms*, Springer, 2004

[37] B. Steffen, G. Levi (eds.), *Verification, Model Checking, and Abstract Interpretation*, Proc. VMCAI 2004, LNCS 2937, Springer 2004, ISBN 3-540-20803-8

[38] W. Menzel, *INCOM – Inputkorrektur durch Constraints und Markups*, E-Learning Consortium Hamburg Projekt, Fachbereich Informatik, Universität Hamburg, 2003

[39] M. Sommer, *Inside CPU*, E-Learning Consortium Hamburg Projekt, Hamburger Universität für Wirtschaft und Politik, 1997

[40] Y. Nahapetian, *Simulation eines von-Neumann-Rechners in der Programmiersprache Java*, Studienarbeit, Fachbereich Informatik, Universität Hamburg, 1996

[41] Lattice Semiconductor Corp., *GAL 16V8 datasheet*, Hillsboro, Oregon, 2004

[42] T. Lindholm, F. Yellin, *Java Virtual Machine Specification, 2nd Edition*, Addison-Wesley, 1999

[43] C. Shannon, *The synthesis of two-terminal switching circuits*, Bell Syst.Techn.J.28, 59-98 (1949)

[44] I. Wegener, *The Complexity of Boolean Functions*, Wiley-Teubner, 1987, http://ls2-www.cs.uni-dortmund.de/monographs/bluebook

[45] Arash Vahidi, *JDD, a Java Binary Decision Diagram Library*, http://javaddlib.sourceforge.net/jdd/intro.html