Statusbericht 02/2004

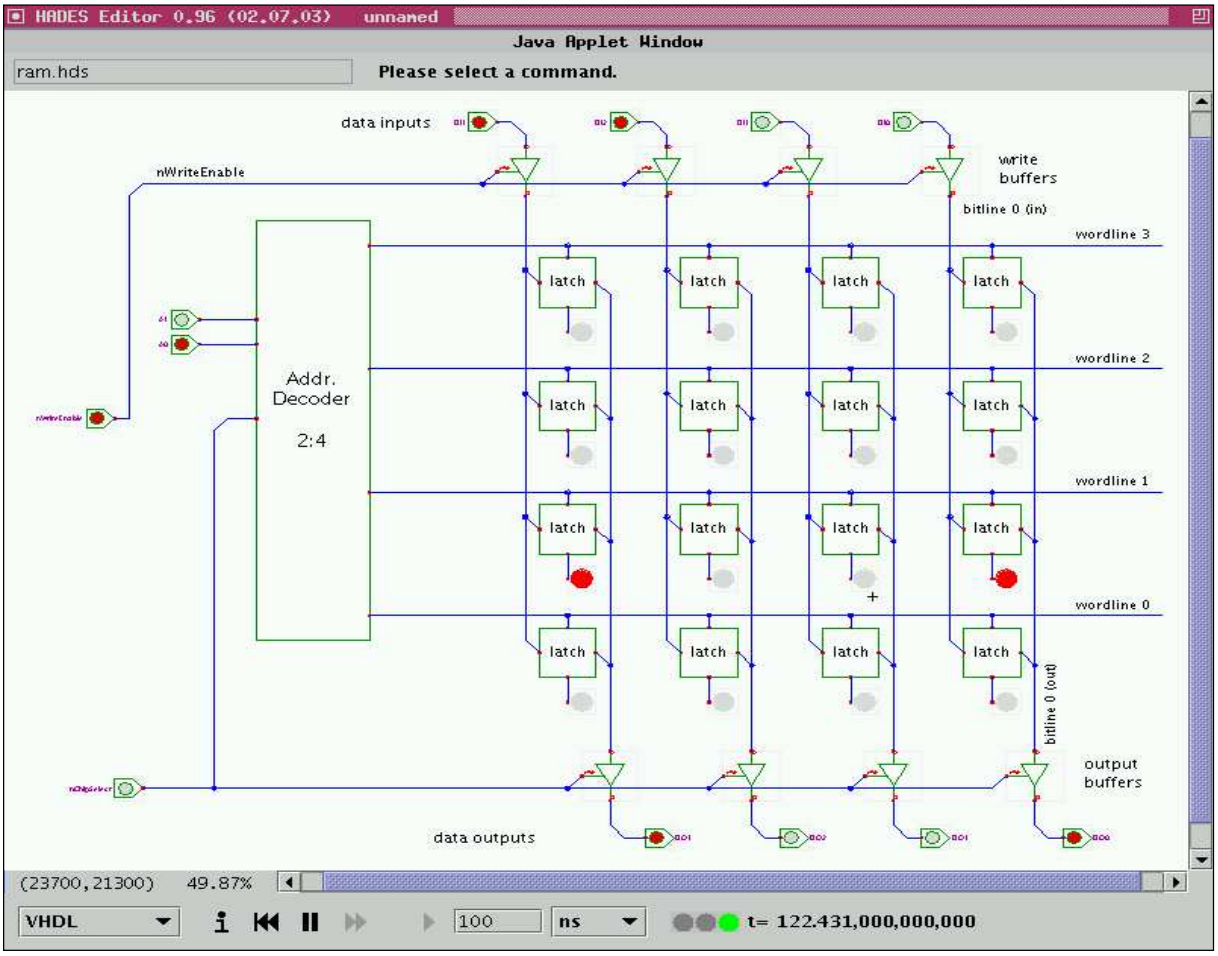
Das interaktive Skript

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

1 Einführung

Der vorliegende, zweite Projektbericht des Projekts „Automatische Überprüfung und Hilfestellung zu Vorlesungs-begleitenden Übungen“ behandelt Fragen der Software-Infrastruktur und erläutert die bisher konkret umgesetzten 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 [10] verwiesen.

und Vertiefung

Als Ergänzung der Beschreibungen wird in den folgenden Kapiteln auch auf kurze Programmbeispiele und Skripte zurückgegriffen, um die zugrundeliegenden Konzepte zu erläutern. Dabei werden die wichtigsten 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* [22] 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.

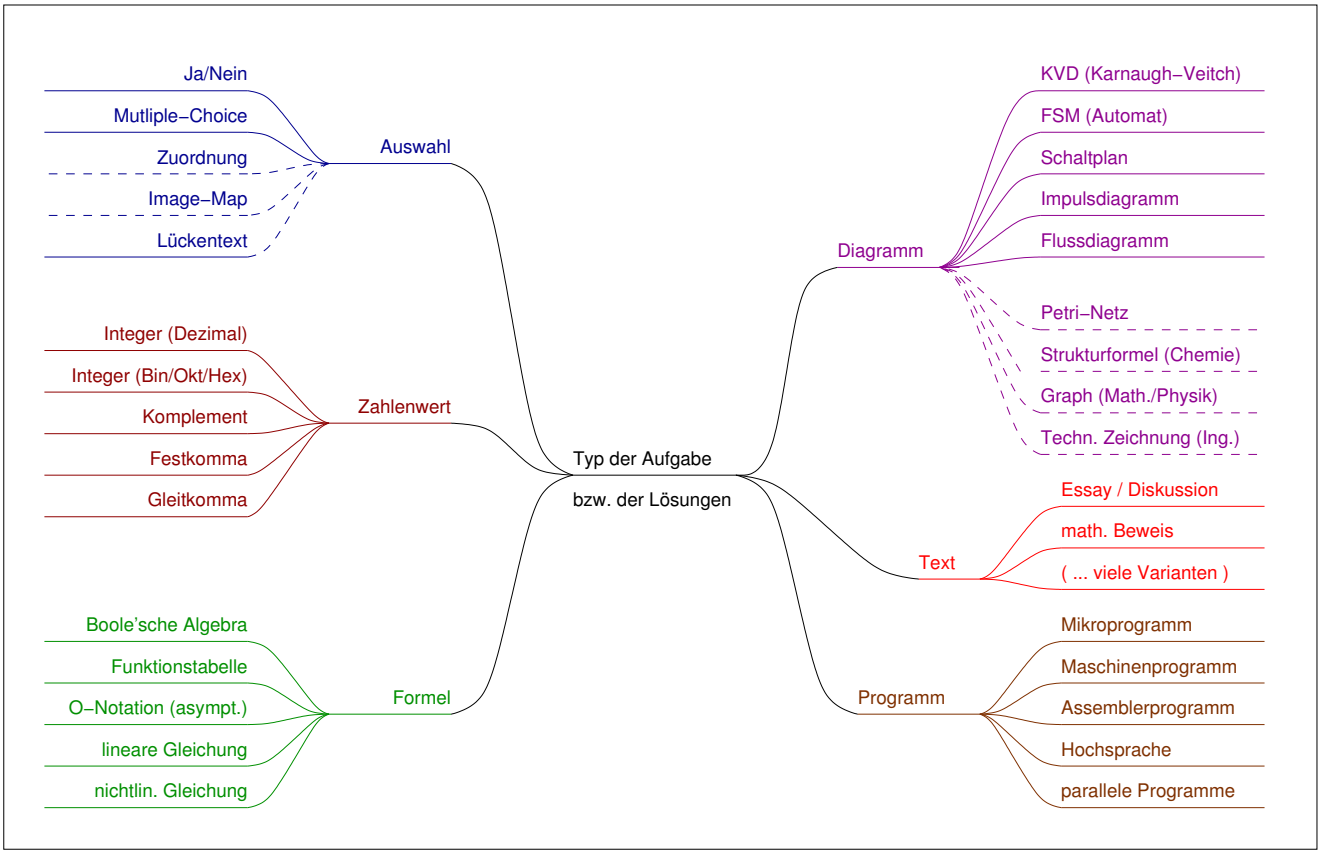


Abbildung 1: *Klassifikation der Übungsaufgaben zur technischen Informatik. Die sechs Hauptklassen dürften auch auf andere mathematisch naturwissenschaftliche Fächer übertragbar sein. Für fast alle Kategorien bis auf freie Texte ist eine automatische Überprüfung möglich.*

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 [10]. 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 begründet und erläutert 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. Dazu werden die bisher entwickelten Hilfs- und Interface-Klassen kurz vorgestellt, wobei auch auf einige kleinere technische Probleme und deren Lösung eingegangen wird. Dies betrifft auch die XML-Repräsentation der Übungsaufgaben und Fragen der Internationalisierung. Anschließend wird gezeigt, wie sich die interaktiven Skripte auch als HTML-Seiten mit kleinen eingebetteten Java-Applets realisieren lassen. 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 *Auswahlaufgaben*, die insbesondere auch die verbreiteten Multiple-Choice- und Zuordnungsaufgaben umfasst. Alternativ zur direkten Realisierung mit Matlab- bzw. Jython-Skripten können diese Aufgabentypen auch mit Hilfe von HTML-Formularen umgesetzt und in die interaktiven Skripte eingebettet werden.

Kapitel 4 Inhalt von Kapitel 4 sind die verschiedenen Formen von *Zahlenwertaufgaben*. Nach einer Übersicht über die verschiedenen Varianten von Aufgaben wird die Klasse *NumberParser* vorgestellt, die nach geeigneter Initialisierung viele dieser Aufgaben überprüfen kann. Danach wird erläutert, wie das direkte Ablesen der Lösungswerte aus dem Skript vermieden werden kann.

Kapitel 5 In Kapitel 5 werden Verfahren vorgestellt, mit denen sich digitale Schaltungen automatisch überprüfen lassen. Dazu wird zunächst das Simulations-Framework Hades vorgestellt, dass als Softwareumgebung zum Bearbeiten von Schaltplänen auf Gatter- und Register-Transfer-Ebene vorgesehen ist. Darauf aufbauend wird gezeigt, wie einfach sich das aus dem Chiptest bekannte Verfahren der LFSR-basierten Signaturanalyse auch zur Überprüfung von Übungsaufgaben zum Schaltungsentwurf einsetzen lässt.

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 eingebetteten Programmcode gestattet. Die von uns gewählte Softwarearchitektur wird in Abschnitt 2.1 vorgestellt. Sie basiert auf einer Kombination der beiden Plattformen *Matlab* und *Java*. Um Probleme mit undokumentierten Datenformaten zu vermeiden, werden alle Nutzdaten (also die Lösungen und Zwischenrechnungen der Studierenden) im Rahmen des Projekts als ASCII-Dateien dargestellt, während für die Metadaten XML eingesetzt wird. Abschnitt 2.2 erläutert dieses Vorgehen und skizziert die Schnittstellen zum Zugriff auf die XML-Daten. Abschnitt 2.3 befasst sich mit dem Problemkreis der Internationalisierung und skizziert Lösungsansätze. Schließlich wird in Abschnitt 2.4 gezeigt, wie sich das Konzept der interaktiven Skripte auch mit den bekannten Standard-Webbrowsern (Internet Explorer, Mozilla, etc.) umsetzen lässt.

2.1 Softwareplattform

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 [18]. 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 so gennannten Toolboxes bzw. Blocksets zusammenfassen. Die Software steht nicht nur für Windows, sondern auch für alle verbreiteten Varianten von Unix zur Verfügung (u.a. Linux, Solaris, MacOS X) und erlaubt damit den plattformunabhängigen Einsatz.

Zwar können Matlab-Skripte über die so genannten *Notebook*-Funktionen direkt in Microsoft-Word Dokumente eingebettet werden, wobei alle Formatierungsmöglichkeiten von Word zur Verfügung stehen und auch eine Schnittstelle zu Excel-Tabellen vorgesehen ist. Leider ist für diese Erweiterung zwingend die Vollversion von Word erforderlich, da andere Office-Software nicht unterstützt wird. Wegen dieser Abhängigkeit und dem Verlust der Plattformunabhängigkeit eignet sich die *Notebook*-Schnittstelle daher weniger für die interaktiven Skripte dieses Projekts.

mscriptview

Als Abhilfe dient *mscriptview*, ein selbstentwickelter Browser, der auf die in Matlab integrierten Funktionen zur Textformatierung zurückgreift und bei Bedarf modular erweitert werden kann. Da Formeln in einer TEX-ähnlichen Syntax direkt in die Skripte integriert werden können, ergibt sich für diesen wichtigen Aspekt gegenüber dem Formeleditor aus Word sogar eine vereinfachte Bedienung. Allerdings reichen die Möglichkeiten zur Formatierung komplexer Formeln bei weitem nicht an die umfangreichen Funktionen von TEX heran.

Lizenzbedingungen

Ein Hindernis für den breiten Einsatz von Matlab in Lehrveranstaltungen und im Kontext von E-Learning bedeuten die Kosten und Lizenzbedingungen, die zudem in der Vergangenheit von MathWorks, Inc. mehrfach geändert worden sind. Eine Vollversion der so genannten Matlab-Suite mit den grundlegenden Komponenten Matlab, Simulink und Symbolic Math Toolbox kostet derzeit immerhin 1400 €.

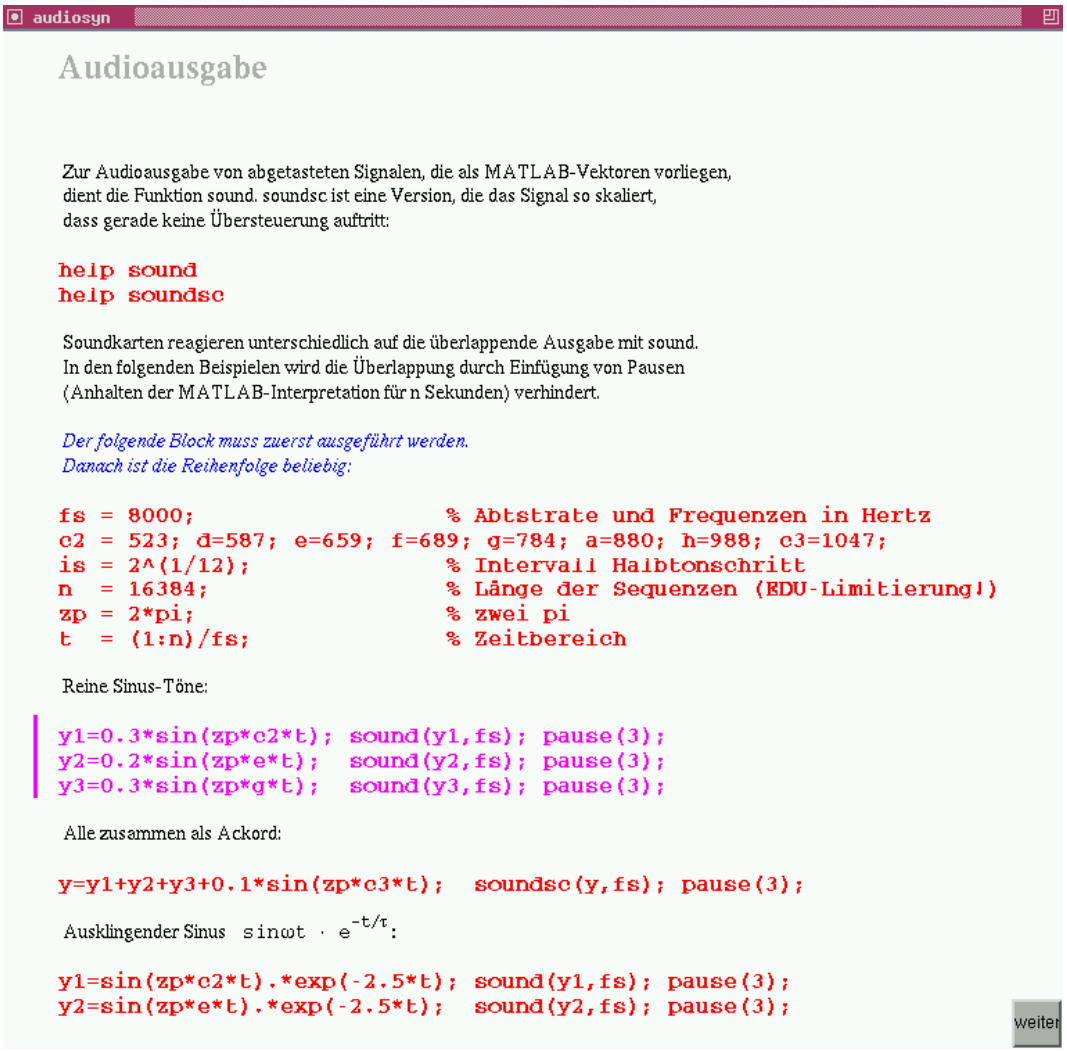


Abbildung 2: *Das interaktive Skript am Beispiel Audiosynthese. Der mscriptview-Browser unterstützt formatierte Texte mit Hyperlinks und Formeln. Zusätzlich sind Programmtexte eingebettet, die beim Anklicken direkt ausgeführt werden; im Beispiel werden Audiodaten berechnet und abgespielt. Der Anwender kann die Skripte jederzeit modifizieren und die Änderungen sofort ausprobieren.*

Student-Version

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 87 € erhältlich, wobei auch hier für zusätzlich benötigte Toolboxes weitere Kosten entstehen. Die immer noch im Abverkauf 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 zu erreichen.

Im Vergleich mit anderen bekannten Softwarepaketen für symbolische und numerische Mathematik (Mathematica, Maple, MuPAD, usw.) bietet Matlab die beste Unterstützung für technische Anwendungen wie Signalverarbeitung, Bildverarbeitung, Datenkommunikation. Dafür mangelt es, gerade im Vergleich mit den umfangrei-

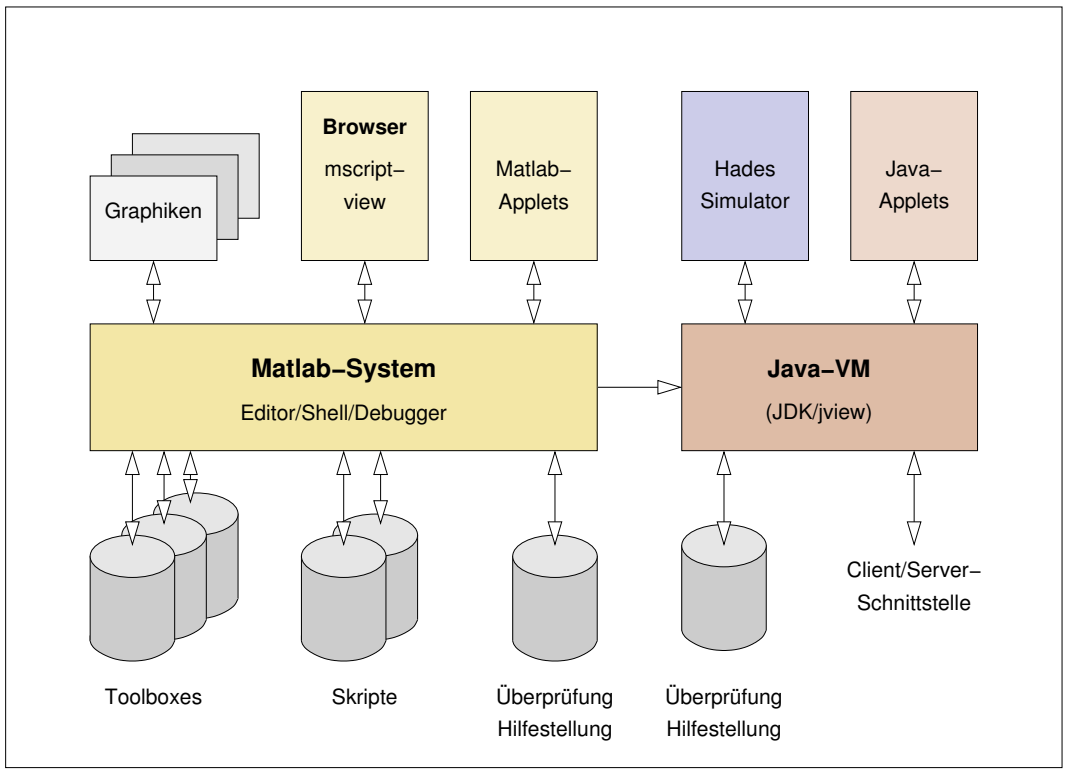


Abbildung 3: *Die Software-Architektur des geplanten Gesamtsystems mit dem Browser zur Darstellung der interaktiven Skripte sowie den Funktions- und Klassenbibliotheken mit Algorithmen zur automatischen Überprüfung der integrierten Übungsaufgaben.*

chen Möglichkeiten der Mathematica *Notebooks*, an Unterstützung zur Darstellung von Formeln und der graphischen Aufbereitung von Dokumenten. Die Symbolic Math Funktionen von Matlab basieren übrigens auf den Algorithmen aus Maple und decken alle Anforderungen dieses Projekts ab.

Java

Neben Matlab wird Java die zweite wichtige Programmierplattform für das Projekt bilden. Die Vorteile von Java wie das durchdachte und saubere Objektkonzept, die umfangreichen Klassenbibliotheken auch von Drittherstellern und die Unterstützung durch eine Vielzahl von Tools sind bekannt und brauchen hier nicht weiter erläutert zu werden. Nicht zuletzt ist Java für alle wesentlichen Desktop- und Serverplattformen frei verfügbar und bietet ideale Voraussetzungen für die Realisierung von Netzwerk- oder Client/Server-Applikationen.

Als klassische compilierte Programmiersprache eigenet sich Java auf der anderen Seite weniger für eine interaktive Umgebung wie das interaktive Skript, da nach jeder Änderung des Quellcodes ein erneutes Compilieren erforderlich ist. Dieses Problem wird dadurch entschärft, dass die aktuellen Java-Entwicklungsplattformen eine inkrementelle Entwicklung erlauben, indem die sehr kurzen Compilerläufe im Hintergrund erfolgen und Klassen während der Entwicklung jederzeit geändert werden können. Zusätzlich existiert für mehrere Skriptsprachen eine Java-Anbindung, die eine interaktive Umgebung für den Zugriff auf alle unter Java verfügbaren Klassenbibliotheken zur Verfügung stellt.

Scripting

Unter anderem gilt dies auch für Matlab selbst, dass seit Version 5.3 eine Java-Schnittstelle beinhaltet, die in Version 6 noch einmal deutlich aufgewertet wurde.

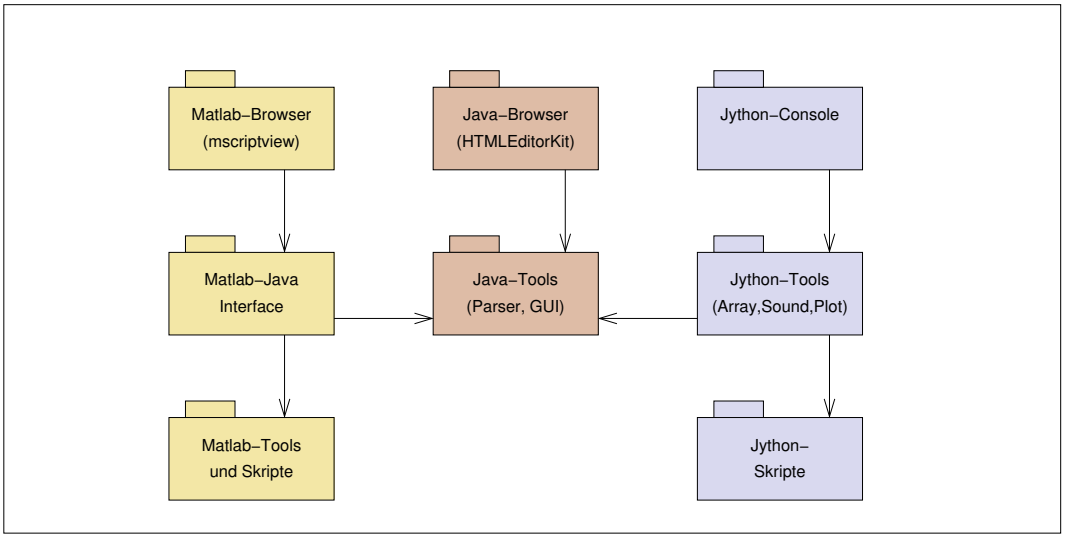


Abbildung 4: *UML Package-Diagramm der Softwarearchitektur des Projekts. Soweit möglich werden die Tools und Parser in Java realisiert, da diese dann von allen beteiligten Plattformen aus nutzbar sind und sich auch für die spätere Integration in E-Learning Plattformen eignen. Die Formel-Überprüfung wird in Matlab realisiert. Die Komponenten auf der rechten Seite beziehen sich auf die als alternative Plattform vorgesehene Kombination aus Java und Jython.*

Dabei kann direkt aus Matlab-Funktionen heraus auf beliebige Java-Klassen und -Methoden zugegriffen werden, wodurch eine Vielzahl bestehender Klassenbibliotheken auch für Matlab zur Verfügung steht. Ein Beispiel bietet die Ansteuerung des von uns in Java entwickelten Hades-Frameworks zur Simulation digitaler Schaltungen aus Matlab heraus, das sich auf diese Weise nahtlos in die interaktiven Skripte zur technischen Informatik einbetten lässt.

Während der Zugriff auf Java-Objekte aus Matlab heraus sehr gut gelöst ist, ist der umgekehrte Weg leider nicht so einfach möglich. Dies bedeutet für das Projekt, dass Matlab sich nur mühsam (auf Umwegen über Objektkommunikation oder temporäre Dateien) als serverbasiertes Hilfsmittel zur Überprüfung der Übungsaufgaben eignet.

Wegen der besonders nahtlosen Integration in das Objektkonzept von Java und einer Reihe von weiteren Vorteilen wird im Projekt außerdem die Sprache Jython [15] für das Skripting von Java-Applikationen eingesetzt werden. Dabei handelt es sich um eine in Java selbst implementierte Variante der modernen, objektorientierten Skriptsprache Python, die frei verfügbar ist und sich ideal zum interaktiven Skripting von Java-Applikationen eignet.

Software-Architektur

Eine Übersicht über die im Projekt geplante Software-Architektur ist in Abbildung 3 dargestellt. Die interaktiven Skripte sind als Matlab-Funktionsbibliotheken realisiert und setzen auf dem Matlab-Kernsystem auf, während die Darstellung und Interaktion über den Browser *mscriptview* erfolgt. Einige der benötigten Funktionen zur automatischen Überprüfung der integrierten Übungsaufgaben werden ebenfalls in Matlab erstellt, zum Beispiel die Auswertung logischer Ausdrücke und die Verarbeitung von symbolischen Funktionen. Andere Komponenten werden in Java erstellt und über die Matlab-Java Schnittstelle angebunden, darunter die Simulation digitaler Schaltungen im Hades-Framework. Die gesamte Software kann problem-

los lokal auf den Rechnern der Studierenden installiert werden, so dass die Skripte auch ohne Netzwerkzugang offline bearbeitet werden. Trotzdem ist eine Client-Server Kommunikation jederzeit möglich, etwa um die Lösungen der fertig bearbeiteten Aufgaben an den Server der Universität zur Bewertung einzusenden.

2.2 Repräsentation der Daten

ASCII,

Wie bereits im ersten Projektbericht ausführlich erläutert wurde, kommen nur portable und wohldefinierte Datenformate zur Repräsentation der Antworten zu den Übungsaufgaben in Frage. Sowohl auf die Texterkennung handschriftlicher Vorlagen als auch auf die komplexen Dateiformate der gängigen Office-Software muss verzichtet werden, da eine ausreichend zuverlässige Erkennung und Verarbeitung dieser Formate nicht garantiert werden kann. Statt dessen werden die eigentlichen Nutzdaten mit den Lösungen der Übungsaufgaben ausschließlich in ASCII- bzw. ISO-8859-1 basierten Textdateien repräsentiert. Dies umfasst auch die Darstellung von Formeln und logischen Ausdrücken, sowie die (optionalen) Nebenrechnungen oder Kommentare der Studierenden zu den Übungsaufgaben.

Quelltexte,

Für ausführbare Skripte, Funktionsdefinitionen, oder kleinere Programme wird auf das Dateiformat der jeweils zugrundeliegenden Softwareplattform zurückgegriffen, zum Beispiel Matlab-Quelltexte oder Java/Jython-Quelltexte. Da diese Quelltexte auf einer Untermenge von ASCII basieren und auf die Parser der zugrundeliegenden Plattform zurückgegriffen werden kann, ist keine zusätzliche Software notwendig.

und XML

Zur automatischen Überprüfung der Übungsaufgaben müssen neben den eigentlichen Nutzdaten — den von den Studenten erstellten Lösungen der Übungsaufgaben — natürlich auch Metainformationen zu den einzelnen Datensätzen verwaltet werden, u.a. die Nummer der Aufgaben und die Namen und Matrikelnummern der Studenten. Angesichts der zunehmenden Verbreitung von auf XML [25] basierenden Dokumentenformaten und Tools bietet es sich an, auch für dieses Projekt die Metadaten in einer XML-Notation zu repräsentieren.

Ein Beispiel für dieses Konzept zeigt Abbildung 5 mit den XML-Verwaltungsinformationen sowie den konkreten Nutzdaten mit Ergebnis und Anmerkung zu einer Übungsaufgabe zum Thema einschrittige Codes. Aus technischen Gründen wird die zugehörige DTD (*Document Type Definition*) derzeit direkt in den einzelnen XML-Dateien abgelegt.

Während das Bearbeiten von XML via Software kein Problem darstellt, ist das manuelle Erstellen und Auslesen von XML-Dokumenten nicht nur sehr mühsam, sondern es kommt auch leicht zu Fehlern. Um wohlgeformte Dokumente sicherzustellen und einen einfachen Zugriff auf die XML-Daten zu gewährleisten, wurden deshalb die notwendigen Werkzeuge implementiert, um die Dateien zu erzeugen und zu parsen. Da nur wenige XML-Tags unterstützt werden müssen, konnte einfach auf den *Crimson*-Parser zurückgegriffen werden, der seit JDk 1.3 zur Java-Standardbibliothek gehört und damit auch in aktuellen Versionen von Matlab zur Verfügung steht. Bei Bedarf lässt sich der Parser aber als zusätzliche Bibliothek auch zusammen mit älteren Versionen von Java verwenden.

Im Rahmen des Projekts stellt die neuentwickelte Hilfsklasse *XmlBuilder* alle Funktionen bereit, um XML-Dateien zu erzeugen. Als Eingabe werden dabei Textdateien

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE aufgabe [
 <!-- DTD für Automatische Überpruefung und Hilfestellung -->
 <!--     von Übungen. Uni-Hamburg / MMKH, 2004           -->
 <!ELEMENT aufgabe (student, zwischenrechnung*, ergebnis*) >
 <!ELEMENT student EMPTY>
 <!ATTLIST student
  name CDATA #REQUIRED
  matrikelnummer CDATA #REQUIRED
>
<!ELEMENT ZWISCHENRECHNUNG (#PCDATA)>
<!ELEMENT ERGEBNIS  (#PCDATA)>
] >

<!-- generated by de.mmkh.tams.XmlBuilder Feb 29 15:01:28 -->
<aufgabe nummer="T1_2_5" >
<student name="Hugo Mustermann"  matrikelnummer="8765432" />
<zwischenrechnung>
Warum akzeptiert der Parser nur Binärzahlen?
Die Aufgabenstellung würde doch durchaus auch Dezimal oder
Hex kodierte Codewörter zulassen.
</zwischenrechnung>
<ergebnis>
0000 0100 1100 1000 1001 1011 1010 1110 0110 0010 0011 0001
</ergebnis>
</aufgabe>
```

Abbildung 5: *Beispiel für die Repräsentation einer Übungsaufgabe. Die Verwaltungsinformationen sind als XML formuliert; aus technischen Gründen ist die DTD direkt in der Datei abgelegt. Die Nutzdaten mit Endergebnis und Zwischenrechnung bzw. Kommentaren zur Aufgabe liegen als normaler ASCII-Text vor.*

mit Ergebnis, Zwischenrechnung und Kommentaren zu einer Übungsaufgabe sowie den Verwaltungsinformationen wie Name und Matrikelnummer des Studierenden erwartet:

```
xb = de.mmkh.tams.XmlBuilder
xb.setStudentName( 'Hugo Mustermann' )
xb.setStudentID( '8765432' )
xb.setExerciseNR( 'T1_2_5' )
xb.setAnswer( '0000 0100 1100 1000 1001 1011' ...
              '1010 1110 0110 0010 0011 0001' )
xb.saveAs( 't1_2_5.xml' )
```

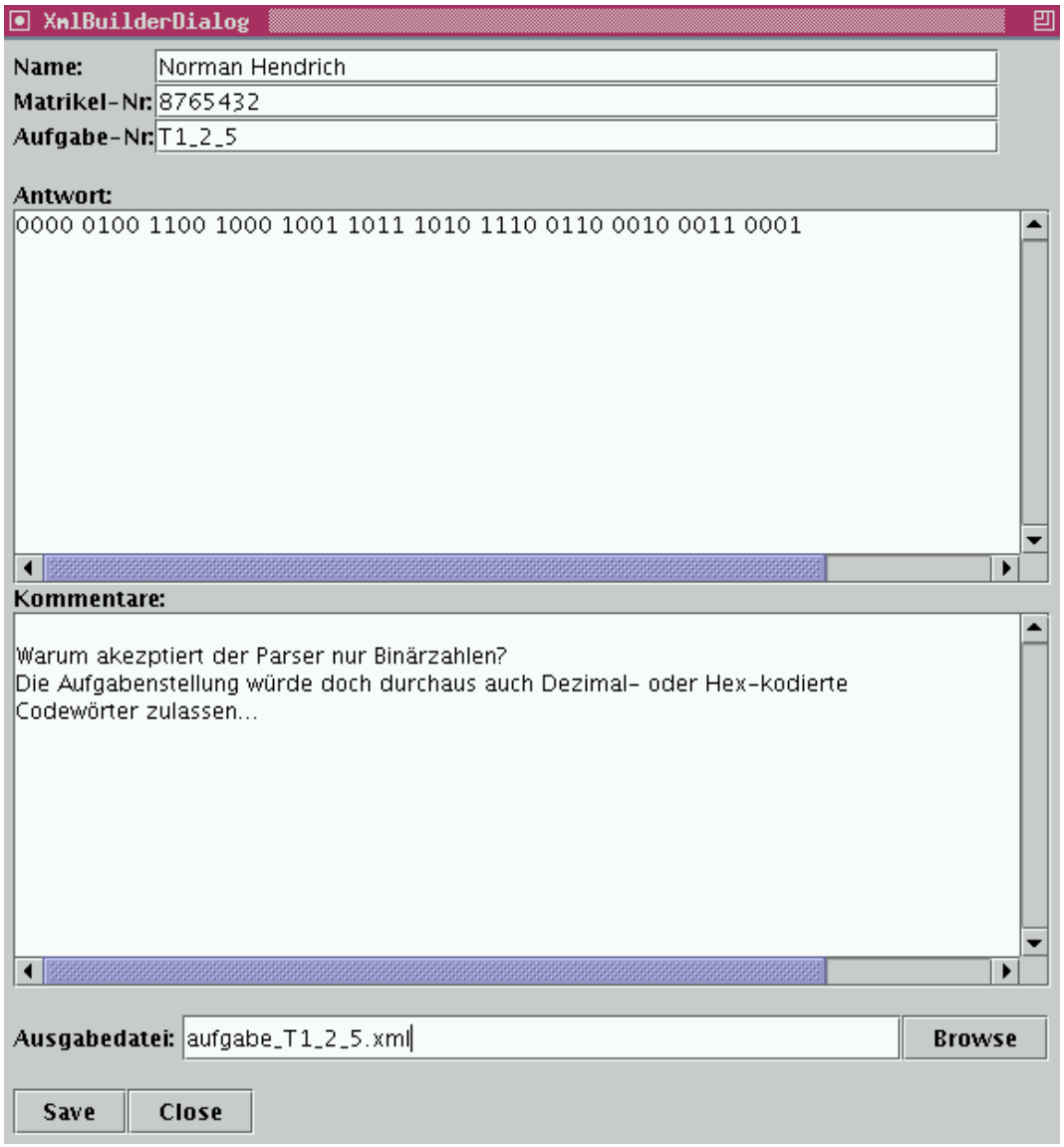


Abbildung 6: *XmlBuilderDialog: Eingabefenster zur einfachen Erstellung der intern verwendeten XML-Dateien. Alternativ ist es natürlich auch möglich, die XML-Dateien programmgesteuert ohne User-Interface direkt aus den interaktiven Skripten zu generieren.*

Analog enthält *XmlParser* die Funktionen zum Auslesen dieser Funktionen aus einer XML-Datei:

```
parser = de.mmkh.tams.XmlParser
parser.parse( 't1_2_5.xml' )
id     = parser.getStudentID
answer = parser.getAnswer
...
```

Schließlich bietet die Klasse *XmlBuilderDialog* ein graphisches Frontend zu *XmlBuilder*, mit dem sich die Nutzdaten interaktiv eingeben lassen. Das User-Interface ist in Abbildung 6 dargestellt.

2.3 Internationalisierung

Das i18n-Problem

Nicht nur im Hinblick auf eine spätere kommerzielle Nutzung der im Projekt entwickelten Software ist es notwendig, von Anfang an den Problemkreis der *Lokalisierung* bzw. *Internationalisierung* („i18n“) zu berücksichtigen. Zwar können für den geplanten Einsatz im Grundstudium der technischen Informatik ausreichende Englischkenntnisse vorausgesetzt werden, wie dies auch in anderen Fachbereichen im universitären Umfeld üblich ist. Damit wäre es möglich, sich als Sprache für die Ein- und Ausgaben des Systems ausschließlich auf Englisch zu beschränken und die Probleme einer Anpassung einfach auszusparen. Dies gilt umso mehr, als die meisten Fachbegriffe in der Praxis nur in ihren englischen Versionen vorkommen.

Auf der anderen Seite zeigt sich, dass viele Studierende die deutschen Versionen der gängigen Lehrbücher gegenüber den Originalen bevorzugen — trotz teilweise erheblicher Qualitätsmängel der Übersetzung. Falls entsprechende Englischkenntnisse nicht vorausgesetzt werden können, ist eine *i18n*-Anpassung zwingend notwendig, um nicht viele Anwender schon durch die Sprachbarriere von der Nutzung des Systems auszuschließen. Es ist daher wünschenswert, eine Anpassung der Software an verschiedene Sprachen zumindest vorzubereiten und durch entsprechende Maßnahmen von vornherein zu erleichtern.

Resource-Bundle

Im Prinzip enthält die Java Standardbibliothek alle Funktionen, die für eine vollständige Internationalisierung benötigt werden. Die Klasse *ResourceBundle* erlaubt einen einfachen Zugriff auf statische Ressourcen wie Strings oder Java-Objekte abhängig von der aktuellen Spracheinstellung (*Locale*). Dabei übernimmt *ResourceBundle* sowohl die Suche und Auswahl der am besten passenden Ressourcen (etwa *de_CH* für Schweizerdeutsch) als auch das Ausweichen auf die Default-Ressourcen (üblicherweise *en_US*). Die konkrete Umsetzung erfolgt durch Installation separater Dateien nach einer bestimmten Namenskonvention, wobei die Sprachversionen als Namenskürzel an den eigentlichen Dateinamen angehängt werden, etwa *NumberParser_de_CH.messages*:

```
NumberParser.messages
# default (English language, United States)
NO_INPUT      = "No input or empty input."
CORRECT       = "The answer is correct."
WRONG_BASE    = "The answer uses a wrong number base."
TOO_BIG       = "The number is too big."
...
```

```
NumberParser_de_CH.messages
# Deutsch, Schweiz
NO_INPUT     = "Die Eingabe ist leer."
CORRECT      = "Die Zahl ist korrekt!"
WRONG_BASE   = "Die Zahl verwendet die falsche Zahlenbasis."
TOO_BIG      = "Die Zahl ist zu gross."
...
```

```
NumberParser_de_DE.messages
...

NumberParser_fr_CH.messages
...
```

Der Zugriff auf die einzelnen Ressourcen erfolgt dann über den entsprechenden Schlüssel, wobei abhängig von der aktuellen Spracheinstellung der zugehörige Wert gesucht und zurückgeliefert wird. Das folgende Beispiel illustriert diesen Mechanismus:

```
// Vorbereitung: einmaliges Laden der Ressourcen
Locale locale = Locale.getDefault(); // z.B. de_DE
ResourceBundle bundle
              = ResourceBundle.
                 getBundle( "NumberParser", locale );
...
String hardkodiert = "The number is too big.";
String i18n        = bundle.get( "TOO_BIG" );
...
```

Es wird deutlich, dass selbst in diesem einfachen Fall ein deutlicher Mehraufwand gegenüber „hardkodierten“ Textausgaben erforderlich ist, da alle Ausgaben über Schlüssel referenziert werden. Zusätzlich müssen alle entsprechenden Übersetzungen in den einzelnen Ressource-Dateien vorliegen und es ist ein gründlicher Test notwendig, um die Konsistenz zu sichern.

Über die verschiedenen Methoden der Klasse *MessageFormat* unterstützt Java auch kompliziertere Fälle, bei denen die Textausgaben aus mehreren einzelnen Komponenten zusammengesetzt werden müssen, etwa um bei Ausgabe eines Zahlenwerts oder Datums zwischen Singular und Plural unterscheiden zu können:

```
MessageFormat format
  = new MessageFormat("The disk \"{1}\" contains {0}.");
double[] limits = {0,1,2};
String[] parts  = {"no files","one file","{0,number} files"};

ChoiceFormat cformat = new ChoiceFormat(limits, parts);
format.setFormatByArgumentIndex(0, cformat);

Object[] testArgs = {new Long(1273), "MyDisk"};
System.out.println(format.format(testArgs));

// output, with different testArgs
output: The disk "MyDisk" contains no files.
output: The disk "MyDisk" contains one file.
output: The disk "MyDisk" contains 1,273 files.
```

Da die für ein *MessageFormat* verwendete Formatierung als String angegeben wird, kann sie bei Bedarf über einen vorherigen Aufruf von *ResourceBundle* abhängig von der Spracheinstellung ausgewählt werden. Damit sind alle Möglichkeiten zur perfekten Lokalisierung von Applikationen gegeben, allerdings um den Preis einer beträchtlichen Komplexität.

Im Rahmen des Projekts wird deshalb zunächst ein Kompromiss zwischen Zukunfssicherheit und Arbeitsaufwand gewählt, indem für alle Textausgaben innerhalb der Java-Klassen bereits symbolische Konstanten verwendet werden. Sofern die Unterstützung zusätzlicher Sprachen notwendig wird, können diese Konstanten dann direkt als Schlüssel für die entsprechenden Aufrufe von *ResourceBundle* und *MessageFormat* dienen. Zusätzlich sind natürlich die entsprechenden Ressource-Dateien zu erstellen.

2.4 Alternative Plattformen

Die in Abschnitt 2 skizzierte Softwarearchitektur basiert im wesentlichen 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 von Matlab selbst eine 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 der Skripte immer 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 Einbettung der interaktiven Skripte in normale HTML-Dateien. Dies würde die Darstellung der Skripte mit einem gewöhnlichen Web-Browser wie dem Internet Explorer oder Mozilla ermöglichen, wobei alle Zusatzfunktionen der Browser zur Verfügung stehen und der Anwender seine gewohnte Umgebung vorfindet. Auch die Integration in bestehende Web-Anwendungen oder E-Learning Frameworks würde deutlich erleichtert. Einige Vorteile der Verwendung eines HTML-Browsers sind offensichtlich:

Vorteile...

- Die interaktiven Skripte können auch sauber angezeigt werden, wenn Matlab nicht zur Verfügung steht — dann allerdings passiv ohne die Interaktionsmöglichkeiten.
- 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, Internet-Recherche.
- Aufruf von Plugins oder externer Hilfsapplikationen wie etwa Postscript- oder PDF-Viewern.

- Bookmarkverwaltung.
- Zugriff auf Sicherheitsfunktionen wie verschlüsselte Datenübertragung, Verwalten von Benutzerpasswörten, Überprüfung digital signierter Inhalte.
- 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.

... und Nachteile

Auf der anderen Seite sind auch einige Nachteile zu verzeichnen. 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 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. Da das Interaktionskonzept zunächst nur auf dem einfachen Anklicken der Codeschnipsel basiert, könnte dies im Prinzip sogar mit HTML-Formularen (Buttons) und entsprechender HTML-Formatierung umgesetzt werden. Da der Web-Browser nach Abschicken der Formulardaten allerdings eine neue Webseite erwartet, wäre eine serverbasierte Lösung notwendig, die neben dem Ausführen des angeklickten Codeschnipsels auch noch die Webseite selbst wieder zurückliefert. Zusätzliche Funktionen, etwa das Markieren des zuletzt angeklickten Codes wären auf diese Weise recht mühsam zu realisieren.

Applet-Restriktionen

Auf der anderen Seite ist eine Implementierung der interaktiven Skripte direkt innerhalb des Web-Browsers aus mehreren Gründen nicht möglich. Wegen des erforderlichen Funktionsumfangs scheidet eine Realisierung mit JavaScript oder etwa als Flash-Plugin von vornherein aus; aus Gründen der Portabilität bleibt damit nur die Umsetzung mit Java-Applets, die aber mit den sehr restriktiven Sicherheitsmechanismen zu kämpfen haben. Insbesondere werden alle auf einer HTML-Seite vorhandenen Java-Applets gestoppt, wenn diese HTML-Seite verlassen wird, so dass auch alle von den Applets erzeugten Fenster oder Daten verloren gehen. Schließlich ist von Java aus derzeit kein Zugriff auf Matlab möglich, so dass nur die Jython-Version der interaktiven Skripte auf diesem Weg realisiert werden könnte.

... und die Lösung

Eine Lösung für dieses Problem bietet die in Abbildung 7 skizzierte Softwarearchitektur. Dabei werden die aktiven Elemente der interaktiven Skripte in unserer aktuellen Referenzimplementierung als kleine Java-Applets (*MatlabApplet* und *JythonApplet*) direkt in die HTML-Seite eingebettet. Die einzelnen Applets selbst übernehmen jedoch nur einfache Aufgaben wie die Behandlung der Mausklicks

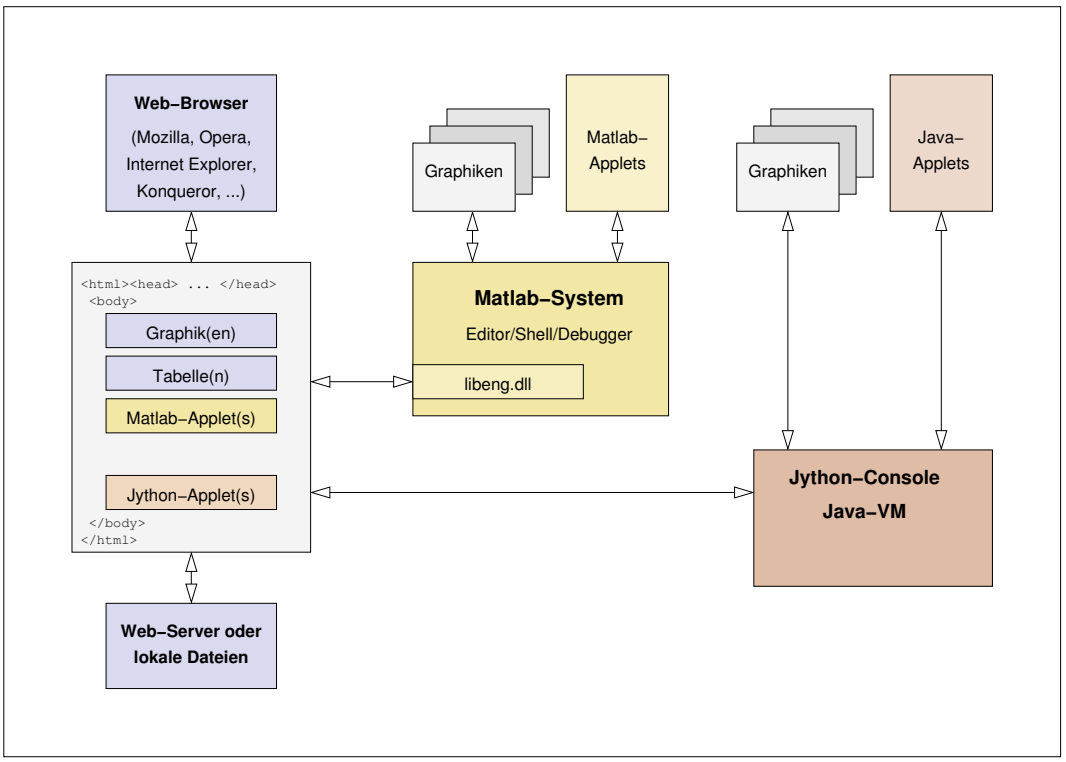


Abbildung 7: *Software-Architektur der HTML-basierten interaktiven Skripte. Die Inhalte werden als HTML-formatierte Webseiten aufbereitet und können mit jedem gewöhnlichen Browser angezeigt werden. Die aktiven Elemente sind als einfache Java-Applets realisiert und kommunizieren mit dem als externe Applikation laufenden Matlab-System bzw. dem Jython-Interpreter.*

und das Hervorheben des zuletzt markierten Applets. Beim Anklicken eines Applets baut dieses eine Netzwerkverbindung zu einem vorher extern gestarteten Serverprozess auf und überträgt seine Nutzdaten, zum Beispiel einige Zeilen Matlab-Programmtext. Die eigentliche Funktionalität der interaktiven Softwareumgebung inklusive der Ausführung der von den Applets übermittelten Programmtexte wird also von extern gestarteten Applikationen übernommen. Da alle Variablen und Funktionen der interaktiven Skripte nicht in den einzelnen Applets sondern der extern gestarteten Softwareumgebung vorgehalten werden, bleiben diese Daten auch beim Wechsel der HTML-Seiten oder sogar zwischen Neustarts des Browsers erhalten.

Konkret implementiert sind derzeit die Anbindung an das Matlab-System sowie an unsere Jython-Console. Der Zugriff auf Matlab erfolgt dabei aus technischen Gründen über die gut dokumentierte C-Schnittstelle (*libeng.dll*), die sogar zusammen mit der älteren Student-Edition (Matalb 5.3) eingesetzt werden kann. Da die Applets nur einfache Textmeldungen mit der Serverapplikation austauschen, lässt sich dieses Prinzip aber auch auf viele weitere Softwareumgebungen (z.B. Mathematica, Datenbanken, usw.) erweitern, sofern diese über eine geeignete Java- oder C-Schnittstelle verfügen.

Remote-Zugriff

Übrigens ließe sich auf diesem Wege natürlich auch ein Zugriff auf die Matlab-Umgebung realisieren, wenn eine lokale Installation von Matlab zum Beispiel wegen der Lizenzkosten nicht möglich ist. Beim gleichzeitigen Zugriff durch mehrere Anwender wäre allerdings zu berücksichtigen, dass Matlab ohne weitere Maß-

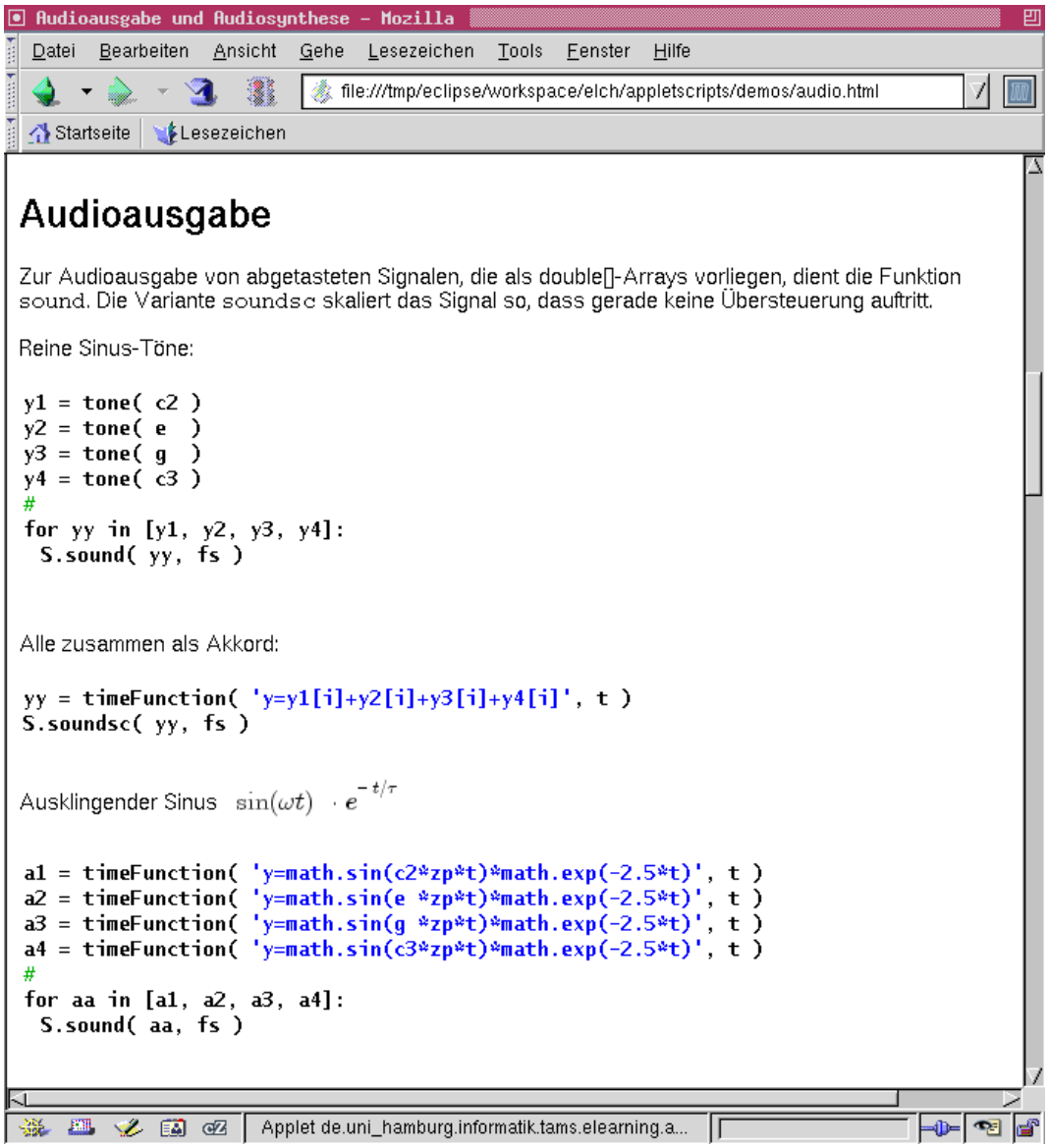


Abbildung 8: *Alternative Plattform: die Jython/HTML-Version des interaktiven Skripts verwendet HTML als Dokumentenformat und kann mit jedem Java-fähigen Browser angezeigt werden. Anstelle von Matlab-Skripten dienen Jython-Skripte als interaktive Elemente. Neben statischen Abbildungen können auch Vektorgraphiken oder LaTeX-formatierte Formeln direkt in den Text eingebettet werden.*
Die im Beispiel gezeigten Funktionen berechnen kurze Audioclips aus den angegebenen Formeln und spielen die Klänge anschließend ab. Anders als in konventionellen Umgebungen ist es jederzeit möglich, die im Skript vordefinierten Funktionen zu modifizieren und durch eigene Experimente zu ersetzen.

nahmen allen Anwendern dieselbe Ausführungsumgebung zur Verfügung stellt. Es bleibt zu klären, ob und wie mehrere Benutzer durch Starten paralleler Threads und Tricks mit Namensräumen gegeneinander abgeschottet werden können.

Ein Beispiel für die Darstellung der interaktiven Skripte in einem Web-Browser liefert Abbildung 8. Hier dient Mozilla 1.6 unter Linux als Benutzeroberfläche für die Jython-Skripte mit den interaktiven Audio-Demonstrationen. Wie oben beschrieben übernehmen die einzelnen Applets nur die Darstellung der Quelltexte inklusive Syntax-Highlighting, während die Berechnung der Audiosignale und die Audioaus-

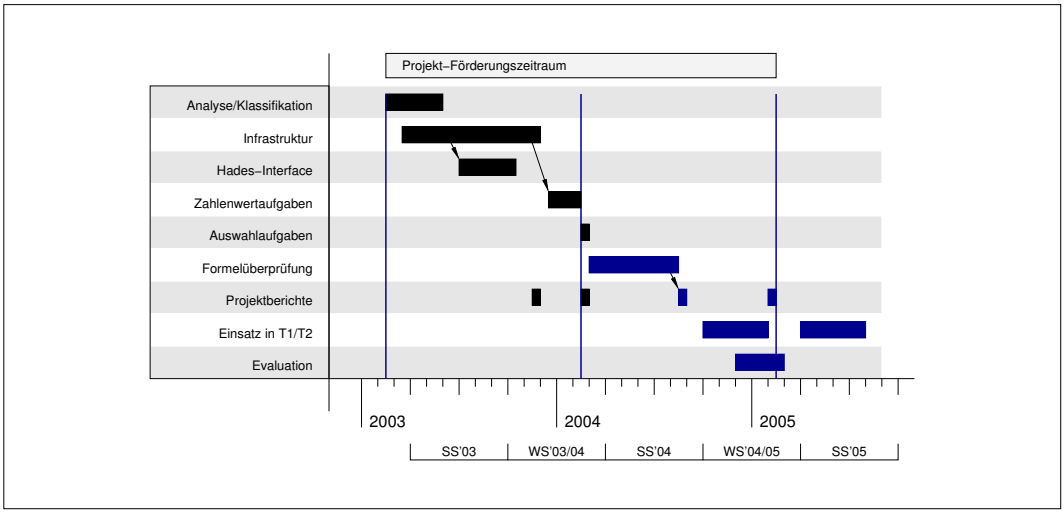


Abbildung 9: *Gantt-Chart zum Projektablauf. Nächstes Ziel ist die Implementierung der Formelüberprüfung innerhalb der Matlab-Plattform. Einsatz und erste Evaluationsergebnisse werden während der T1-Vorlesung im WS'04/05 erwartet.*

gabe von der Jython-Console übernommen werden.

Pure-Java Browser

Falls kein geeigneter Web-Browser zur Verfügung steht oder wegen Sicherheitsbedenken Applets nicht aktiviert werden dürfen, kann im Notfall auf einen selbstentwickelten Browser zur Darstellung der interaktiven Skripte zurückgegriffen werden. Dieser Browser basiert auf einer aufgebohrten Version der Java-Klasse *HTMLEditorKit*, die einen vollwertigen Parser für HTML 3.2 enthält, der lediglich um die notwendige Unterstützung von Applets erweitert wurde. Ein Screenshot dieses Browsers ist in Abbildung 10 auf Seite 20 dargestellt. Da es sich um einen Prototyp handelt, wurden bisher nur die Grundfunktionen zum Öffnen und zur Darstellung der HTML-Seiten implementiert.

Mobilgeräte

Trotzdem eignet sich dieser Browser auch als Grundlage für ein weiteres und in Zukunft vermutlich zunehmend wichtiges Szenario, nämlich die Verwendung der interaktiven Skripte auf mobilen Geräten wie PDAs oder SmartPhones. Tatsächlich dürften Geräte der aktuelle Generation PocketPC-basierter Organizer bereits genügend Rechenleistung und Speicher bereitstellen, um die interaktiven Skripte darstellen zu können. Während es derzeit aber noch keine Matlab-Version für PocketPCs gibt, ist Java für diese Plattform bereits verfügbar und auch Jython kann im Prinzip verwendet werden. Nur die geringe Bildschirmauflösung der Geräte (derzeit typischerweise mit 320x240 und maximal 320x480 Bildpunkten) bedeutet noch eine erhebliche Einschränkung.

2.5 Zeitplan

Zum Abschluß ist in Abbildung 9 die aktuelle Zeitplanung zum weiteren Ablauf des Projekts skizziert. Zentrale Aufgabe während des Sommersemesters 2004 ist die Implementierung der verschiedenen Werkzeuge zur Überprüfung von Formeln inklusive logischer Ausdrücke. Parallel dazu müssen Teile der Projekt-Infrastruktur vorbereitet und getestet werden, um den praktischen Einsatz im Rahmen der Vorlesung Technische Informatik T1 im WS'2004/2005 zu gewährleisten.

3 Auswahl-Aufgaben

Wegen der Typenvielfalt der möglichen Aufgabenstellungen und diverser Probleme bei der konkreten Implementierung werden aber üblicherweise nur einige wenige, besonders einfach umzusetzende Arten von Aufgaben zugelassen. Als Beispiel zählt die Kurzbeschreibung der auch im Rahmen des ELCH-Projekts verfügbaren Lernplattform CLiX [24] die folgenden vom System unterstützten Typen von Übungsaufgaben auf:

- Ja/Nein-Fragen und Multiple-Choice-Fragen
- Zuordnungs-Fragen, Reihenfolgen
- Numerische Zahlenwerte
- Lückentexte
- Image-Maps
- Fließtext

All diesen Aufgabentypen bis auf die Kategorie Fließtext ist gemeinsam, dass die Antwort sofort durch einen trivialen Vergleich mit der korrekten Musterlösung überprüft werden kann. Bei den Multiple-Choice und Zuordnungs-Fragen reicht es vollständig aus, die Antwort als richtig oder falsch einzuordnen; eine feinere Einteilung in Zwischenwerte oder die Bewertung von fast richtigen Teillösungen ist kaum notwendig.

Ganz anders sieht es in der Kategorie Fließtext aus, bei der die Unterstützung durch das Framework sich auf das reine Abspeichern der von den Studierenden eingegebenen Texte beschränkt. Da die Lernplattform keine über den Datentransfer hinausgehenden Funktionen bereitstellt, muss die Durchsicht und Korrektur wie üblich durch die Lehrpersonen erfolgen.

Maßnahmen zur automatischen Hilfestellung bei falschen Antworten zu Lösungen der ersten Kategorien sind selten; üblich dagegen ist es, die weitere Navigation innerhalb des Lehrmaterials solange einzuschränken, bis die im Material eingestreuten Übungen korrekt gelöst sind. Es ist im Einzelfall abzuwägen, ob diese Behinderung wirklich zu einem gründlicheren und erneuten Durcharbeiten des Stoffes führt — oder zu wildem Durchprobieren der angebotenen Multiple-Choice Antworten. Zumindest aus Sicht der Autoren kann eine derartige Beschränkung der Navigationsmöglichkeiten leicht Frust auslösen und die weitere Beschäftigung mit dem Material erschweren.

3.1 Hilfestellungen

Typisch für alle Arten von Auswahlaufgaben ist die geringe Anzahl der möglichen Lösungen und der geringe Informationsgehalt der einzelnen Lösungen. Jede Ja-Nein-Auswahl bedeutet offenbar ein einzelnes Bit, und auch Multiple-Choice Fragen mit Mehrfachauswahl umfassen jeweils nur wenige Bits. Es ist daher so gut wie unmöglich, aus einer falschen Lösung auf die zugrundeliegende inhaltliche Ursache zurückzuschließen um daraus wiederum eine geeignete Hilfestellung abzuleiten. Einfache Konsistenzchecks sind natürlich trotzdem möglich, so dass sich zumindest „formelle“ und Flüchtigkeitsfehler vermeiden lassen. So lässt sich zum

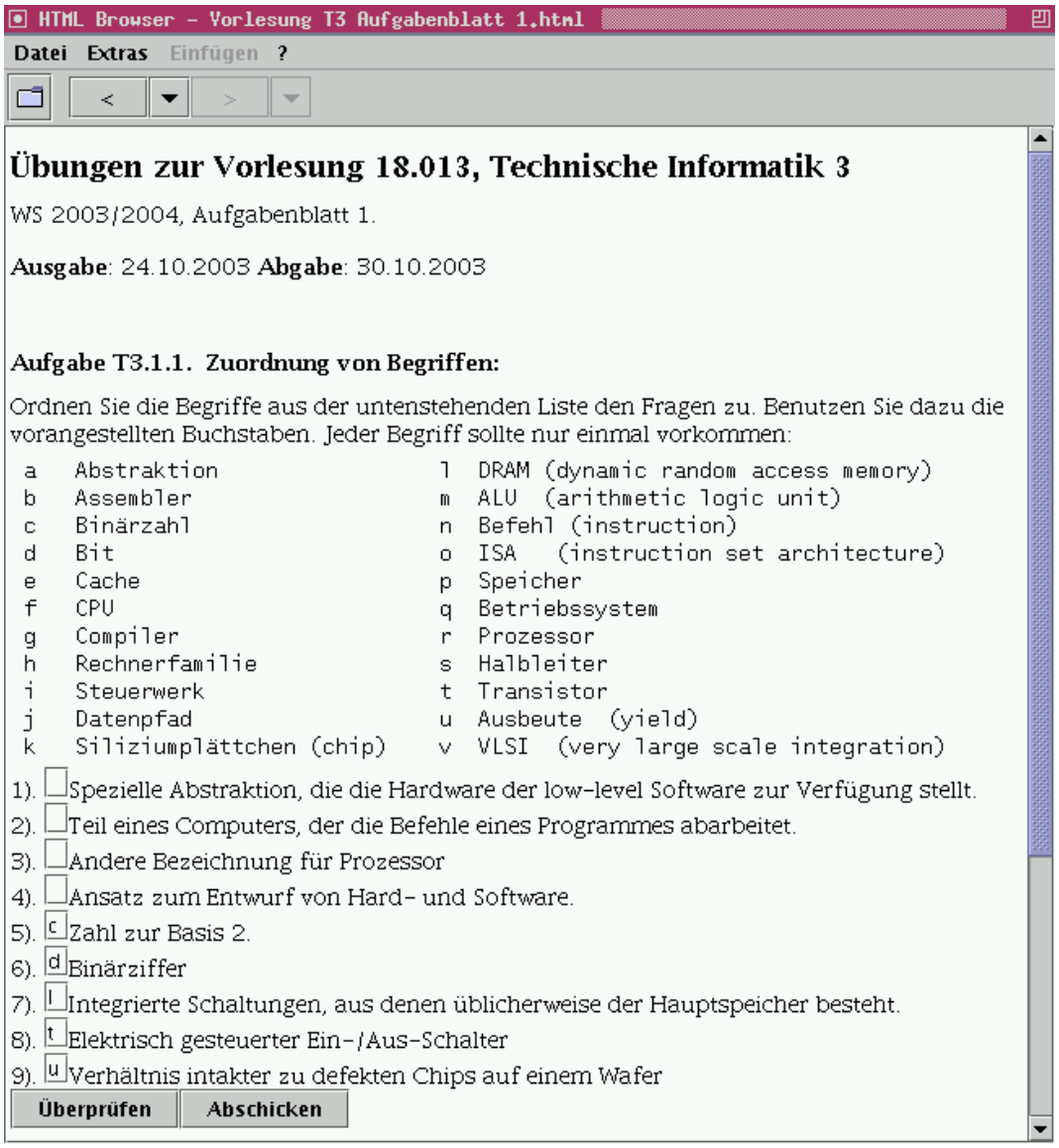


Abbildung 10: *Ein Beispiel für eine Zuordnungsaufgabe aus der Vorlesung T3. Die Eingaben in den Textfeldern werden intern als Strings verwaltet und ausgewertet. Gleichzeitig zeigt die Abbildung den Prototyp eines in Java realisierten Browsers für die interaktiven Skripte. Anders als unser Matlab-basierter Browser mscriptview und die bekannten Web-Browser (vgl. Abbildung 8) kann dieser Browser auf allen J2ME-kompatiblen Plattformen eingesetzt werden. Mit Abstrichen in Bezug auf Performance und Bildschirmauflösung ist damit auch auf Mobilgeräten wie PDAs ein Zugriff auf die interaktiven Skripte möglich.*

Beispiel überprüfen, ob die Mindest- bzw. Höchstanzahl der anzukreuzenden Optionen auch wirklich ausgefüllt wurde, oder ob bei Zuordnungs-Aufgaben Einträge mehrfach verwendet wurden. Bei Lückentexten ergeben sich eventuell zusätzliche Möglichkeiten, etwa das Vorschlagen von „Eselsbrücken“ anstelle der korrekten Lösung.

3.2 HTML-Formulare

Für die Einbettung der verschiedenen Typen von Auswahl-Aufgaben in ein HTML-basiertes Framework wie CLiX werden üblicherweise HTML-Formulare [29] verwendet, die alle Grundelemente der gängigen User-Interface Toolkits wie Buttons, RadioButtons, Auswahllisten, ein- und mehrzeilige Textfelder, etc. umfassen. Als Beispiel zeigt Abbildung 10 die Darstellung einer Zuordnungs-Übungsaufgabe zur Vorlesung Technische Informatik T3. Der Aufgabentext ist als HTML-Tabelle realisiert, während für die einzelnen Eingabefelder normale Textfelder zum Einsatz kommen. Obwohl die Abbildung unseren experimentellen, Java-basierten *E-Learning Browser* zeigt, können entsprechende Webseiten mit jedem gewöhnlichen Web-Browser angezeigt und bearbeitet werden.

Das Interaktionskonzept von HTML-Formularen basiert allerdings immer auf einer Client-Server Kommunikation. Nach Ausfüllen des Formulars wird dieses über Anklicken des entsprechenden Action-Buttons (hier „Überprüfen“ und „Abschicken“) an einen Webserver geschickt und dessen Antwort als nächste Seite dargestellt. Daher ist für dieses Konzept eine Online-Verbindung zum zugehörigen Webserver zwingend notwendig. Bei Bedarf kann ein externer Webserver selbstverständlich durch lokal gestartete Hintergrundprozesse ersetzt werden, so dass die Option für ein ein Offline-Bearbeiten der entsprechend realisierten Aufgaben in den interaktiven Skripten trotzdem offen bleibt.

Alternative JavaScript?

Beim Einsatz eines Web-Browsers zur Darstellung der interaktiven Skripte ergibt sich übrigens eine weitere Möglichkeit, die verschiedenen Typen von Auswahl-Aufgaben in die Skripte einzubetten und zu überprüfen. Dabei werden weiterhin HTML-Formulare zur Darstellung der Aufgaben verwendet, während die Algorithmen zur Überprüfung direkt als JavaScript-Code in die HTML-Seiten integriert werden. Dieses Verfahren funktioniert mit allen Web-Browsern, die JavaScript ausreichend unterstützen (Internet Explorer, Mozilla, Opera, und viele weitere) und erlaubt sowohl die sofortige Überprüfung innerhalb des Browsers selbst als auch die Kommunikation mit einem externen Server.

Diesen Vorteilen stehen aber auch einige schwerwiegende Nachteile gegenüber. Dies betrifft zum einen diverse Inkompatibilitäten der von den verschiedenen Browsern unterstützten Sprachvarianten von JavaScript, was die Erstellung und Pflege der entsprechenden Skripte sehr erschwert. Zweitens sind die Möglichkeiten von JavaScript als Programmiersprache sowohl gegenüber Matlab als auch der Kombination von Jython mit Java doch schon sehr stark eingeschränkt und auf die Umgebung der jeweiligen Webseite beschränkt. Komplexere Algorithmen, etwa zur Überprüfung anspruchsvollerer Aufgaben, lassen sich kaum noch sinnvoll umsetzen. Nicht zuletzt würde der Einsatz von JavaScript für dieses Projekt die Einführung einer weiteren Programmiersprache bedeuten. Schließlich beruhen viele der immer wieder neu entdeckten Sicherheitslücken in den gängigen Browsern auf Scripting-Funktionen. Bis ein Update des Browsers zur Verfügung steht, lassen sich solche Sicherheitslücken dann nur durch komplettes Abschalten von JavaScript schließen.

4 Zahlenwert-Aufgaben

Neben den im vorherigen Kapitel behandelten Auswahlaufgaben sind *Zahlenwertaufgaben* die zweite wichtige Klasse von Übungsaufgaben. Diese Aufgaben kommen in den betrachteten Vorlesungen und Lehrbüchern, anders als die Auswahlaufgaben, sehr häufig und in vielen Variationen vor. Der Lehrstoff der technischen Informatik umfasst dabei auch Aufgabenstellungen, die über die üblichen Grundfunktionen hinausgehen, etwa Radixdarstellungen mit beliebiger Basis.

In Abschnitt 4.1 werden die Konzepte zur Überprüfung und Hilfestellung für diese Klasse von Aufgaben zusammengefasst. Anschließend erläutert Abschnitt 4.2 den im Rahmen des Projekts entwickelten Parser für Zahlenwerte und demonstriert einige Beispielszenarien. Für die Überprüfung ist es erforderlich, die erwarteten korrekten Werte am Parser einzustellen; die gesuchte Lösung der Aufgabe ist also im Skript selbst vorhanden und damit den Studierenden zugänglich. Lösungsmöglichkeiten für dieses Problem werden in den Abschnitten 4.3 und 4.4 diskutiert. Schließlich präsentiert Abschnitt 4.5 eine Beispielfunktion, die gezielt zur Überprüfung einer einzelnen Aufgabe dient, und erläutert die zugehörigen Hilfestellungen.

4.1 Ansätze zur Überprüfung und Hilfestellung

Übersicht

Die Kategorie der Zahlenwertaufgaben umfasst alle Aufgabenstellungen, deren Lösung aus einem oder mehreren Zahlenwerten besteht. Im einfachsten — und häufigsten — Fall ist die korrekte Lösung der Aufgabe eindeutig, so dass die Überprüfung sich auf einen trivialen Vergleich mit der Musterlösung beschränkt. Es ist daher nicht überraschend, dass diese Art der Aufgaben auch von E-Learning Frameworks wie CLiX seit langem unterstützt wird. Wie bereits im ersten Projektbericht angedeutet wurde, treten in vielen Fachgebieten aber auch Aufgabenstellungen mit anwendungsspezifischen zusätzlichen Anforderungen auf. Beispiele sind die Darstellung von sehr großen Zahlen, unkonventionelle Zahlendarstellungen, Komplementdarstellungen, periodische Brüche, oder erhöhte Genauigkeitsanforderungen. Eine wichtige Trennung ergibt sich auch inhaltlich. Während viele mathematisch und theoretische orientierte Fragestellungen eindeutige Integerwerte als Lösung aufweisen, treten Gleitkommawerte häufig als Lösungen naturwissenschaftlicher Probleme auf. Dabei sind die Ausgangswerte der Aufgabenstellung häufig bereits mit Fehlern behaftet, die sich bis in die endgültigen Lösungswerte fortpflanzen. Trotzdem ist auch für diese Typen von Aufgaben die Überprüfung im Prinzip immer noch mit einem einfachen Wertevergleich mit einer Musterlösung möglich. Weitere Tests erlauben es, die zusätzlichen Attribute wie Zahlenbasis, Stellenanzahl oder Genauigkeit zu überprüfen.

Elementare Hilfestellungen vs. Diagnose

Ähnlich wie bei den Auswahlaufgaben ist der Informationsgehalt eines einzelnen Zahlenwerts zu klein, um bei Fehlern wirklich gute Hilfestellungen zu erzeugen. Natürlich ist es einfach, kurze Meldungen in der Art von „Der Wert ist zu klein“ oder „Das Vorzeichen ist falsch“ zu generieren, aber darüberhinausgehende Diagnosen sind kaum möglich. Insbesondere kann aus der der Eingabe eines falschen Zahlenwerts nur in Ausnahmefällen auf die zugrundeliegende Fehlerursache zurückgeschlossen werden, etwa wenn der zur Lösung der Aufgabe notwendige Algorithmus

eindeutig vorgegeben ist. Ein Beispiel dafür ist die Umrechnung von Dezimalzahlen in die Binärdarstellung mit dem zugehörigen sehr einfachen Algorithmus. Bei der Umrechnung kann es jedoch leicht zu Flüchtigkeitsfehlern wie Vertauschungen oder ausgelassenen Rechenschritten kommen, die sich in diesem Fall durch typische Fehler der Resultwerte bemerkbar machen. Nicht zuletzt aus diesem Grund umfasst unsere XML-Repräsentation der Lösungen optional auch Einträge mit Nebenrechnungen und Anmerkungen, auch wenn die Auswertung im Fehlerfall in den meisten Fällen wohl weiterhin den Übungsgruppenleitern vorbehalten ist. In jedem Fall erhalten die Studierenden durch die sofortige Fehlermeldung die Chance, die Rechnung solange zu wiederholen, bis das Ergebnis stimmt.

Toleranzintervalle

Für Aufgaben mit Gleitkommazahlen ergeben sich zusätzliche Anforderungen, um Rundungsfehler geeignet zu behandeln. Schon das Vertauschen einiger arithmetischer Operationen in einem ansonsten korrekten Lösungansatz kann leicht zu Ergebnissen führen, bei denen der direkte Vergleich mit der vorgegebenen Musterlösung fehlschlägt. Zudem ergeben sich Gleitkommazahlen häufig als Resultat komplexer Aufgabenstellungen, bei denen vereinfachende Annahmen oder Schätzungen mit in den Lösungsweg eingehen. Eventuell lassen sich die Anforderungen sogar mit vollkommen verschiedenen Architekturen und Lösungsvarianten erfüllen. Es ist daher notwendig, nicht nur einen erwarteten bitgenauen Wert als korrekte Lösung zu akzeptieren, sondern alle Werte innerhalb eines bestimmten, vorher eingestellten Toleranzintervalls.

Einheitenvergleich

Die gerade bei Anwendungen in den Naturwissenschaften ebenfalls wünschenswerte Unterstützung von Zahlenwerten in Kombination mit den zugehörigen Einheiten (etwa $9.81\,\mathrm{m/s^2}$ anstelle von 9.81) ist zwar als Option vorgesehen, wurde jedoch bisher noch nicht umgesetzt. Eine Realisierung könnte auf den Klassen des *Java Units* Pakets aufbauen, das derzeit als Erweiterung der Java Standardbibliothek diskutiert wird. Eine Referenzimplementierung dieser Klassen liegt bereits vor [30] und dürfte sich bei Bedarf leicht in das Projekt integrieren lassen:

```
JADE.initialize();              // einmalige Initialisierung

Unit unit1 = SI.JOULE.divide( SI.METER );
Unit unit2 = unit1.mulitply( NonSI.HOUR );
Unit unit3 = Unit.valueOf( "1 Kg/m/s2*rad" );

Quantity q = Quantity.valueOf( 100, unit3 );
...
```

Da fehlende Einheiten in vielen Fällen einen Rückschluss auf falsche Lösungsansätze oder vergessene Rechenschritte erlauben, könnte ein Einheitenvergleich auch dazu dienen, bessere Fehlermeldungen bzw. Hilfestellungen zu generieren. Nicht zuletzt könnten die gängigen Zehnerpotenzen bei der Auswertung der numerischen Werte berücksichtigt werden. Ein Beispiel ist die je nach Anwendungsfall durchaus sinnvolle Angabe der Lichtgeschwindigkeit in den Einheiten 300.000 km/s bzw. 30 cm/ns. Bis zur Integration der Java-Units Klassen ist es als Zwischenlösung immerhin möglich, die Zahlenwerte und die zugehörigen Einheiten getrennt voneinander auszuwerten. Bereits ein einfacher Stringvergleich der Einheiten der von den Studierenden erstellten Lösung mit der Musterlösung könnte dabei helfen, Flüchtigkeitsfehler zu entdecken.

4.2 NumberParser

Trotz der oben angedeuteten Varianten kann die eigentliche Überprüfung von Zahlenwertaufgaben letztlich immer auf den direkten Vergleich mit einer Musterlösung zurückgeführt werden. Im Rahmen des Projekts wurde dazu die Klasse *NumberParser* realisiert, die die gemeinsamen Grundfunktionen kapselt und auch die elementaren Hilfestellungen bereitstellt. Zur Überprüfung einer Aufgabe werden zunächst im zugehörigen Skript die Parameter für Zahlenformat und den erwarteten Lösungswert einmal eingestellt und anschließend mit der Eingabe der Studierenden verglichen. Die derzeitige Version unterstützt die folgenden Zahlenformate:

- Integerzahlen (Dezimalzahlen). Die Java-basierte Repräsentation erlaubt hier die Zweierkomplementdarstellung mit 32- bzw. 64-bit Wortbreite. Die größte darstellbare Zahl ist also 9.223.372.036.854.775.807, was für die meisten Anwendungen ausreichen dürfte.
- Integerzahlen mit vorgegebener Zahlenbasis, insbesondere Binär-, Oktal- und Hexadezimalzahlen. Als Schreibweise wird die Zahlenbasis am Ende des Zahlenwerts angehängt (z.B. `babe_16`). Für Binär- und Hexadezimalzahlen wird auch die aus C und Java bekannte Notation `0b0011` bzw. `0xcafe` erkannt.
- Dezimalbrüche.
- Gleitkommazahlen im IEEE-754 Format (double precision).

Erweiterungen

Eine Unterstützung von Brüchen zu anderer Zahlenbasis ist bisher nicht extra implementiert, da dies leicht durch getrennte Eingabe der Vor- und Nachkommastellen und einzelne Überprüfung ersetzt werden kann. Aufgrund der intern verwendeten Java-Klassen kann eine Zahlenbasis b aus dem Intervall $2 \leq b \leq 36$ verwendet werden. Wie üblich werden die Zahlen bei einer Basis von $b > 10$ mit den Dezimalziffern und den Buchstaben `0123456789abcdef...xyz` kodiert.

Ebenfalls noch nicht integriert ist die Unterstützung von sehr großen Integerzahlen (mehr als 64 bit), wie sie zum Beispiel in der Kryptographie oder bestimmten kaufmännischen Problemen auftreten können. Bei Bedarf ist jedoch problemlos möglich, passende Funktionen zu ergänzen, die auf die Klassen *BigInteger* bzw. *BigDecimal* aus der Java-Standardbibliothek zurückgreifen.

Vor der Überprüfung eines Zahlenwerts werden die erwarteten Attribute der Lösung im Parser eingestellt; nicht explizit gesetzte Werte werden bei der Überprüfung ignoriert. Konkret verfügt NumberParser Einstellungen für:

- das Zahlenformat: Integer, Gleitkomma, Bruch
- den erwarteteten eigentlichen Zahlenwert
- ein Toleranzintervall für den Wert (insbesondere für Gleitkommazahlen).
- die Zahlenbasis
- die Anzahl der erwarteten Ziffern

API

Die Programmierschnittstelle zu NumberParser besteht aus einer Reihe von *set*-Methoden, um die gewünschten Parameter einstellen zu können. Die meisten Methoden liegen in zwei Versionen vor, um sowohl numerische Werte als auch Zeichenketten als Argumente verwenden zu können. Anders als in der objektorientierten Programmierung üblich sind keine zugehörigen *get*-Methoden vorhanden, damit die einmal eingestellten Werte nicht direkt wieder ausgelesen werden können:

```
parser    = de.mmkh.tams.NumberParser;    % Parser erzeugen
parser.setExpectedFormat( 'INTEGER' );    % Ganzzahl
parser.setExpectedBase( 16 );             % Basis 16 ("hex")
parser.setExpectedNumberOfDigits( 4 );    % genau 4 Stellen
parser.setExpectedTolerance( 0 );         % exakte Lösung
parser.setExpectedValue( 51966 );         % Lösungswert 'cafe'

eingabe   = input;                        % Eingabe abwarten
status    = parser.check( eingabe );      % und überprüfen
```

Beispiele

Als Beispiel für die Reaktion von *NumberParser* mit den obigen Einstellung enthält das folgende Protokoll die Antworten („status“) für eine Anzahl von Benutzereingaben. Die jeweils erste Zeile enthält die Eingabe, die untere Zeile die zugehörige Statusmeldung der Überprüfung:

```
0xcafe
Die Lösung ist korrekt.

0xaffe
Der Wert ist zu klein.

0x0000cafe
Falsche Anzahl der Stellen.

0xcafg
Die Zahl enthält ungültige Ziffern.

cafe_16
Die Lösung ist korrekt.

cafe_15
Die Zahlenbasis stimmt nicht.

51966
Die Zahlenbasis stimmt nicht.

3.14159265
Die Eingabe ist keine Integerzahl.

Elchtest!
Bitte eine gültige Zahl eingeben.
```

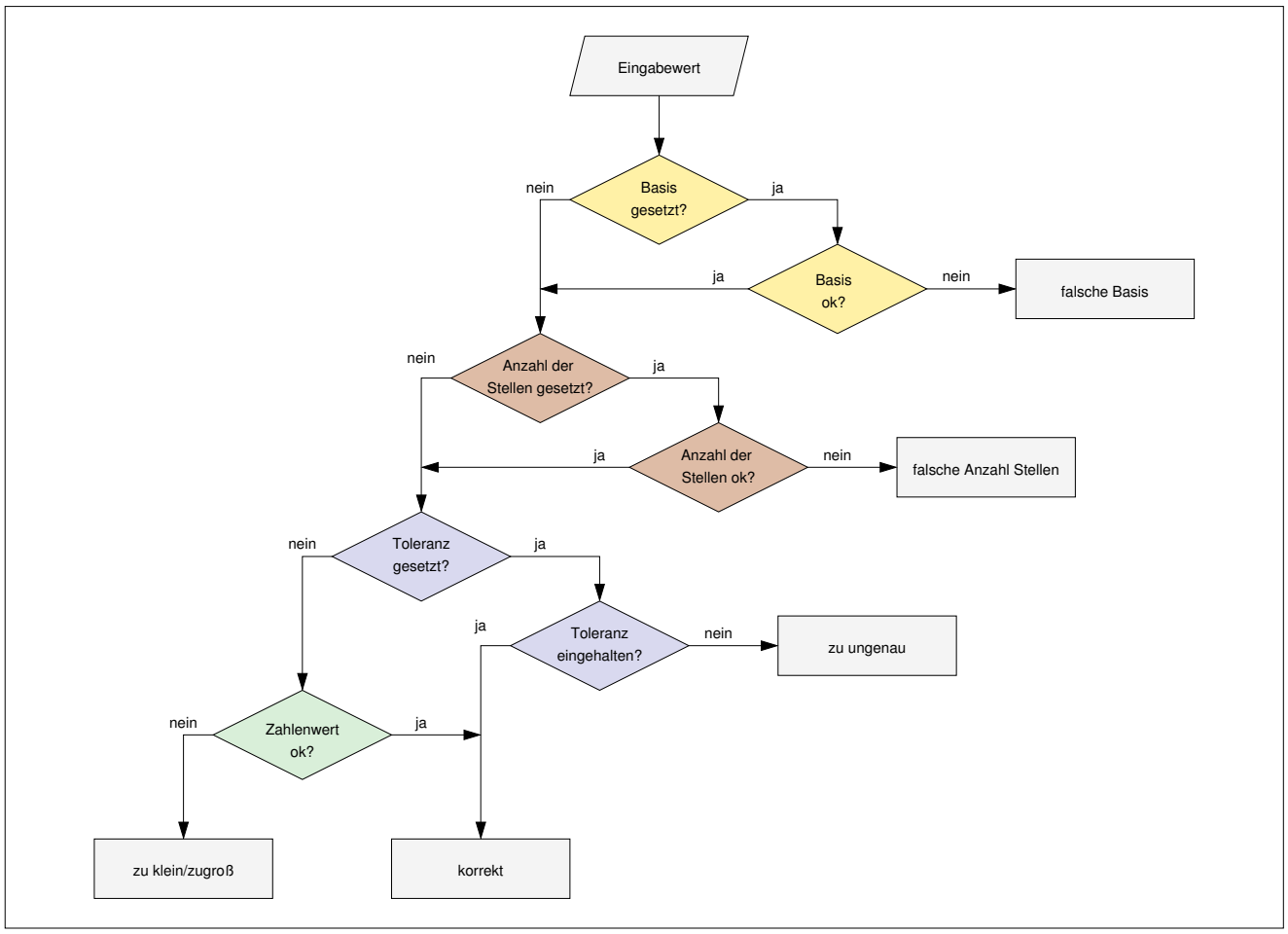


Abbildung 11: *Flußdiagramm zur Überprüfung von Integerzahlen. Nicht gesetzte Attribute werden bei der Überprüfung übersprungen.*

Die Antworten von *NumberParser* beschränken sich daher nicht nur auf ein einfaches „richtig“ und „falsch“, sondern umfassen auch eine Reihe der oben als „elementar“ bezeichneten Hilfestellungen. Das intern zugrundeliegende Prinzip der Überprüfung ist in Abbildung 11 als (vereinfachtes) Flußdiagramm am Beispiel der Integerzahlen dargestellt. Falls einzelne Attribute der Lösung, etwa die genaue Anzahl der Ziffern, nicht explizit gesetzt wurden, werden die entsprechenden Tests übersprungen. Die Abfolge der einzelnen Abfragen hat wesentlichen Einfluß auf die Ausgaben, die im Fehlerfall generiert werden. Die gezeigte Reihenfolge weist der Zahlenbasis die größte Rolle zu — sollte zum Beispiel für die obige Eingabe `51966` zusätzlich noch die Meldung erfolgen, dass der Zahlenwert selbst schon korrekt ist?! Hier werden die Erfahrungen zeigen, ob die gewählte Reihenfolge akzeptabel ist oder noch umgestellt werden sollte.

4.3 Das Klartext-Problem

Lösung direkt im Skript...

An dieser Stelle wird ein Problem deutlich: die notwendigen Funktionsaufrufe zur Einstellung der erwarteten Attribute enthalten bereits die Lösung der Aufgabe im Klartext. Sie dürfen also nicht direkt in ein interaktives Skript aufgenommen werden, da sonst die Lösung bereits in der Aufgabenstellung selbst vollständig vorweggenommen wird und von den Studierenden lediglich abgetippt werden muss. Es ist klar, dass dies die Motivation zur ernsthaften Bearbeitung der Aufgaben nicht unbedingt steigert.

... in externen Funktionen...

Es ist daher notwendig, die Einstellungen von NumberParser in externe Funktionen zu verlagern, die dann wiederum aus dem interaktiven Skript heraus aufgerufen werden. Es bleibt aber das Problem, dass die Anwender der interaktiven Skripte — zumindest im bisher von uns verfolgten Konzept — über den vollständigen Quelltext aller vom Skript referenzierten Funktionen verfügen. Der Aufwand zum „Nachschlagen“ der Lösung beschränkt sich also auf das Öffnen der entsprechenden Funktion im Editor.

... oder compiliert

Als Alternative bietet es sich an, die entsprechenden Funktionen nicht im Quelltext sondern nur in einer vorcompilierten Version bereitzustellen. Leider ist auch dieses Vorgehen nicht vollständig sicher, da es immer noch möglich ist, den Programmcode mit einem Debugger oder Decompiler zu analysieren. Dies kann zwar in den Lizenzbedingungen der Software ausgeschlossen werden, ein Verstoß lässt sich jedoch nur schwer ermitteln. Darüber hinaus ist es bei Java-Klassen aufgrund der enthaltenen Metadaten fast immer möglich, den ursprünglichen Quelltext vollständig zu rekonstruieren, während compilierter C- oder Matlab-Code zumindest einen gewissen Schutz bietet.

Überprüfung auf dem Server

Damit bleibt als einzige brauchbare Lösung für dieses Problem die Verwendung einer Client-Server Architektur, bei der die Daten der Lösungen ausschließlich auf dem Server gehalten werden und den Anwendern niemals zur Verfügung stehen. Statt dessen senden die Studierenden ihre Lösungen (als XML-Dateien) zum Server, der die Auswertung übernimmt und nur den entsprechenden Status und eventuelle Hilfestellungen zurückliefert, aber keine darüber hinausgehenden Informationen. Leider erfordert dieser Ansatz eine Online-Verbindung zur Überprüfung, die mit den entsprechenden Kosten und dem Nachteil der eventuell beträchtlichen Antwortzeiten verbunden ist.

Signaturen an Stelle der Lösungswerte

Bei einigen Typen von Aufgaben ist es glücklicherweise möglich, aus der korrekten Lösung eine digitale Signatur oder einen Hashwert zu berechnen, der dann anstelle der Lösung zur Überprüfung dienen kann. Neben den bekannten kryptographisch sicheren Hashfunktionen wie MD5 oder SHA, die sogar für digitale Signaturen ausreichen, kommen eventuell auch anwendungsspezifische einfachere Funktionen in Frage. Ein derartiges Verfahren ist die Signaturanalyse digitaler Schaltungen, die in Abschnitt 5.3 ab Seite 38 vorgestellt wird.

4.4 NumberScrambler und StringScrambler

Verschleierung

Angesichts der obigen Diskussion wird deutlich, dass ein Kompromiss zwischen leichter Lösung und leichter Bedienbarkeit der Übungsaufgaben gefunden werden muss. Hierzu dient die Java-Klasse *NumberScrambler*, die ihre Eingabedaten mit einigen arithmetischen und logischen Operationen so umrechnet, dass die ursprünglichen Werte nicht mehr leicht zu erkennen sind. Anschließend werden dann diese verschleierten Werte aus den Skripten an *NumberParser* übergeben, so dass die Originalwerte nicht mehr benötigt werden:

```
% Übungsaufgabe mit Lösung im Klartext im Skript
%
parser    = de.mmkh.tams.NumberParser;     % Parser erzeugen
parser.setExpectedValue( 42 );             % Lösungswert
eingabe   = input;                         % Eingabe abwarten
status    = parser.check( eingabe );       % und überprüfen

% Verschleierte Lösung erzeugen - später nicht im Skript:
%
mask      = 987654321                      % Zufallswert
scrambler = de.mmkh.tams.NumberScrambler   % Generator
mask, tmp = scrambler.scramble( mask, 42 ) % Verschleiern
% liefert '3ade68b1' '197921dc'

% Übungsaufgabe mit kodierter Lösung, um das direkte
% Ablesen zu erschweren:
%
parser    = de.mmkh.tams.NumberParser;
parser.setExpectedValued( '3ade68b1', '197921dc' )
eingabe   = input;
status    = parser.check( eingabe );
```

Die Funktion zur Entschlüsselung ist dabei direkt in *NumberParser* integriert und von außen nicht zugreifbar. Trotzdem ist das *NumberScrambler* implementierte Verfahren keinesfalls kryptographisch sicher; es dient aber auch lediglich dazu, die Hemmschwelle zum direkten Nachschlagen der Parameter soweit zu erhöhen, dass das Bearbeiten der eigentlichen Übungsaufgabe einfacher erscheint.

StringScrambler

Natürlich betrifft das Problem der Klartext-Darstellung nicht nur numerische Werte, sondern alle Arten von Zeichenketten ganz allgemein. Dies bedeutet, dass auch für alle anderen Typen von Aufgaben bei Bedarf die Kodierung der Musterlösungen bzw. Signaturen unlesbar gemacht werden muss. Wiederum bietet die Client-Server Architektur die beste Sicherheit. Anders als bei Zahlenwerten bietet compilierter Programmcode in Hinblick auf Zeichenketten übrigens keinen ernsthaften Schutz, da die in Programmen und Binärdateien enthaltenen Zeichenketten mit leicht bedienbaren Werkzeugen angezeigt werden können. Unter Unix leistet dies zum Beispiel das *strings* Utility. Entsprechende Funktionen sind auch Bestandteil der gängigen Debugger, so dass kaum eine Abhilfe gegen ernsthafte Entschlüsselungsversuche durch die Anwender besteht. Neben dem Ausweichen auf die Client-Server

Variante bleibt nur die Alternative, die Musterlösungen nicht als Strings sondern in einem geeignet gewählten Binärformat zu kodieren.

Im Rahmen der interaktiven Skripte und zur Auswertung von Übungsaufgaben reicht allerdings wiederum ein einfaches Verfahren aus, um das direkte Ablesen der Lösung zu erschweren. Daher wurde eine Klasse *StringScrambler* implementiert, die das gleiche Prinzip wie *NumberScrambler* benutzt. Das zugrundeliegende Verfahren beruht auf der Base64-Kodierung [28], wobei die Reihenfolge des zugrundeliegenden Alphabets aber noch frei gewählt werden kann, so dass ein Entschlüsseln mit vorhandenen Base64-Werkzeugen erschwert ist. Als Nebeneffekt kann *NumberScrambler* auch genutzt werden, um Zeichenketten mit beliebigen ASCII-Sonderzeichen in ein druckbares und portables Zeichenformat umzusetzen. Ein kleines Matlab-Beispiel reicht aus, um den Einsatz zu illustrieren:

```
% Übungsaufgabe mit Lösung im Klartext im Skript:
%
loesung = '(x1 & x2 & x3) | (x1 & ~x2 & ~x3) | (~x1 & ~x2)'
eingabe = input;
status  = checkPlaintext( eingabe, loesung );

% kodiert, um das direkte Ablesen zu erschweren:
%
kodiert = de.mmkh.tams.StringScrambler( loesung )
kodiert = 'KHgxICYgeDIgJngzKSB8ICh4MSAm' ...
          'IH54MiAmIH54MykgfCAofngxICYgfngyKQ=='
eingabe = input;
status  = checkEncoded( eingabe, kodiert );
```

Durchprobieren aller Lösungen

Da der Wertebereich der möglichen Lösungen bei vielen Aufgaben recht klein ist, könnten einige Studierende bei der Suche nach der richtigen Antwort auch auf die Idee verfallen, kurzerhand alle Lösungen auszuprobieren. Ein derartiges Vorgehen wird natürlich durch die Software-Plattform der interaktiven Skripte ideal unterstützt; eine kleine Schleife reicht aus, um die Lösung vom Skript selbst berechnen zu lassen:

```
for k=1,1000000
  status = check_aufgabe_xyz( k );
  if (status == NumberParser.OK)
    sprintf( s, 'Der gesuchte Wert ist %l', k )
  end
end
```

Anzahl- und Zeitbudget

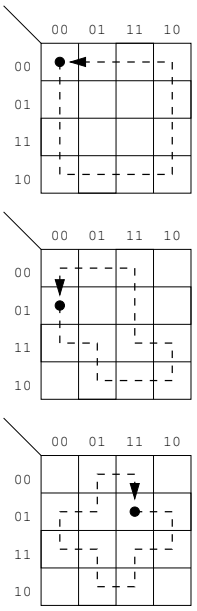
Über die Beherrschung der jeweiligen Programmierumgebung und -sprache hinaus muss aber bereits ein Grundverständnis der Aufgabenstellung vorhanden sein, damit dieses Vorgehen sinnvolle Resultate liefert… Insofern kann ein derartiger Lösungsversuch zumindest teilweise durchaus als Lernerfolg verbucht werden. Eine einfache Abhilfe gegen das Scripting der Algorithmen zur Überprüfung beruht auf dem Mitprotokollieren der Anzahl der Lösungsversuche, eventuell kombiniert mit einem Zeitbudget. Falls mehr als einige Dutzend Anfragen gezählt werden, liegen vermutlich keine manuellen, wohlüberlegten Eingaben vor, und es wird ein Fehlerstatus zurückgeliefert.

4.5 Beispiel: Einschrittiger zyklischer Code

Für bestimmte Aufgabenstellungen wird die Überprüfung nur teilweise mit allgemein nutzbaren Klassen wie *NumberParser* möglich sein, so dass die Entwicklung zusätzlicher Funktionen notwendig ist, die genau auf die jeweilige Aufgabe zugeschnitten sind. In vielen Fällen wird sich der Zusatzaufwand in akzeptablen Grenzen halten, sofern auf geeignete Grundfunktionen zurückgegriffen werden kann. In diesem Abschnitt wird ein konkretes Beispiel für eine derartige, dedizierte Funktion vorgestellt, die zur Überprüfung für genau eine einzelne Übungsaufgabe dient. Als zugrundeliegende Übungsaufgabe dient dabei erneut die Aufgabe T1-2.5 zum Thema einschrittige Codes, die bereits im ersten Projektbericht im Zusammenhang mit der „überprüfungsgerechten“ Formulierung von Aufgabenstellungen diskutiert wurde [10]. Hier noch einmal die Aufgabestellung aus den Übungen zur Vorlesung T1 im WS2002/2003:

> **Aufgabe T1-2.5**: Finden Sie einen zyklisch-einschrittigen Code mit 12 Codewörtern. Ein solcher Code könnte z. B. für eine Winkelcodierung in 30^{o}-Schritten benutzt werden.

Da eine korrekte Lösung zu dieser Aufgabe aus einer Liste mit den zwölf Codewörtern besteht, erscheint die automatische Überprüfung der Lösung trivial durch Vergleich der Codewörter mit einer Musterlösung möglich zu sein. Das Problem mit diesem Ansatz ist jedoch, dass die Aufgabe nicht nur eine einzige sondern vielmehr eine Vielzahl von korrekten Lösungen zulässt, die zwar in Bezug auf die Anforderungen gleichwertig sind, sich aber nicht direkt (etwa durch Umsortieren) ineinander überführen lassen. Als Beispiel sind in der Randspalte drei mögliche Lösungen anhand von KV-Diagrammen skizziert, wobei die Codewörter entlang der Pfeilrichtung abzulesen sind. Offenbar ergeben sich weitere gültige Lösungen, wenn der Pfad in umgekehrter Richtung oder ausgehend von einem anderen Startfeld durchlaufen wird. Es wird aber auch sofort deutlich, dass es viele weitere gültige Wege (und damit Lösungen der Aufgabe) gibt.



Eine vollständige Auflistung aller korrekten Lösungen wäre zwar möglich, ist wegen der großen Anzahl der Lösungen aber kaum praktikabel. Die automatische Überprüfung der Lösung zu dieser Aufgabe dürfte daher die meisten am Markt erhältlichen E-Learning Frameworks überfordern, selbst wenn diese über die nötigen Parser für Binärzahlen verfügen.

Trotzdem ist für diese Aufgabenstellung eine Überprüfung der Lösung auf Korrektheit problemlos mit einem einfachen Algorithmus realisierbar. Dazu wird zunächst getestet, ob die Lösung genau 12 unterschiedliche Codeworte der Länge 4 bit umfasst. Ausgehend vom ersten Codewort wird anschließend überprüft, ob sich die nachfolgenden Codeworte jeweils um nur 1 bit unterscheiden. Auch eine interaktive Hilfestellung ist für diese Aufgabe leicht zu implementieren, da ein passender Assistent die wesentlichen Merkmale der Lösung (Anzahl, Länge und Hamming-Distanz der einzelnen Codewörter) mit geringem Aufwand überprüfen kann. Daraus lassen sich dann wiederum hilfreiche Fehlermeldungen und Tips ableiten.

Der oben skizzierte Algorithmus ist in Abbildung 12 als Jython-Pseudocode dargestellt, wobei die einzelnen Phasen des Algorithmus durch verschiedene Farben hervorgehoben sind. Eine entsprechende Implementierung liegt auch in Java vor; das

folgende Codebeispiel zeigt eine mögliche Einbettung dieser Klasse *Check_T1_2_5* in ein Matlab-basiertes interaktives Skript:

```
parser  = de.mmkh.tams.Check_T1_2_5;  % Parser erzeugen
eingabe = input;                      % Eingabe abwarten
status  = parser.check( eingabe );    % und auswerten
status                                % Status ausgeben
```

Das folgende Protokoll enthält jeweils eine Zeile mit einer Beispieleingabe für den Parser und darunter die zugehörige Ausgabe bzw. Statusmeldung des Parsers:

```
Hallo
Falsche Anzahl der Codewörter

a b c d e f g h i j k l
Das Codewort a ist keine gültige Binärzahl

0 1 2 3 4 5 6 7 8 9 10 11
Das Codewort 2 ist keine gültige Binärzahl

0000 0001 0011 0010 0110 0111 0110 0111 1111 1110 1100 1000
Das Codewort 0110 ist doppelt.

0000 0001 0011 0010 0110 0111 0110 0111 1111 1110 1100
Falsche Anzahl der Codewörter

0000 0101 1100 1000 1001 1011 1010 1110 0110 0010 0011 0001
Hamming-Distanz zwischen Codewörtern 0000 und 0101
ist nicht eins.

0000 0100 1100 1000 1001 1011 1010 1110 0110 0010 0011 0001
Die Lösung ist korrekt

1101 1001 1011 1111 1110 0110 0111 0011 0001 0101 0100 1100
Die Lösung ist korrekt
```

Natürlich erkennt der Parser die erste Eingabe als ungültig; es ist jedoch sehr schwierig, für fehlerhafte Eingaben dieser Art sinnvolle Fehlermeldungen zu generieren. Die Fehlermeldung („falsche Anzahl der Codewörter“) erklärt sich in diesem Fall einfach dadurch, dass die Überprüfung der Anzahl der Codewörter als erster Test durchgeführt wird, während die Auswertung der einzelnen Codewörter erst im nächsten Schritt erfolgt.

Der Versuch, die weiteren Beispiel-Eingaben manuell auszuwerten, zeigt dann aber doch schnell den Nutzen der automatischen Überprüfung, zumal eine einzelne Überprüfung nur Sekundenbruchteile dauert. Sofern ein Anwender des Systems überhaupt eine Idee zur Lösung der Aufgabe hat, erlauben die Fehlermeldung ein schrittweises Herantasten an die korrekte Lösung. Auch Flüchtigkeitsfehler werden zuverlässig erkannt und lassen sich daher leicht vermeiden.

```
def check_T1_2_5( stringWithValues ):
  values = convertToValues( stringWithValues )
  if (len(values) != 12): error( 'falsche Anzahl der Codewörter' )
  for i in range(0, 12):
    if( nbits(values[i]) < 4): error( 'Codewort zu kurz:', values[i] )
  for i in range(0, 12):
    for j in range(0, 12):
      if (i == j) continue
      if (values[i] == values[j]): error( 'Codewort doppelt:', values[i] )
  for i in range(0, 12):
    j = (i+1) mod 12
    if( hamming(values[i],values[j])!=1): error( 'nicht einschrittig' )
  return 'Lösung ist korrekt'
```

Abbildung 12: *Pseudocode zur Überprüfung der Aufgabe T1.2.5. Der Algorithmus zerlegt den Eingabestring in die einzelnen Codewörter und überprüft dann in vier Schritten, ob die Codewörter die in der Aufgabe geforderten Bedingungen einhalten (Anzahl der Codewörter, Länge der einzelnen Codewörter, mehrfach vorkommende Codewörter, Hamming-Distanz der Codewörter). Die tatsächlich implementierte Version verwendet ausführlichere Statusmeldungen (siehe Text).*

5 Überprüfung digitaler Schaltungen

Dieses Kapitel behandelt Verfahren zur Überprüfung von Übungsaufgaben zum Thema Schaltungsentwurf. Die dabei vorkommenden *Schaltpläne* bilden die erste wichtige Gruppe von *Diagramm-Aufgaben* [10].

Nach einer kurzen Übersicht über das Themengebiet in Abschnitt 5.1 beschreibt Abschnitt 5.2 das Prinzip der interaktiven Simulation und die dazu von uns ausgewählte Software. Anschließend erläutert Abschnitt 5.3 das BIST-Verfahren zur automatischen Analyse einer Schaltung mittels Simulation und Signaturanalyse, das sich auch gut zur Überprüfung von Übungsaufgaben eignet. Ergänzende Techniken zur Generierung von Hilfestellungen werden in Abschnitt 5.4 vorgestellt.

5.1 Einführung

Schaltbilder

Ein Themenschwerpunkt im Rahmen des Grundstudiums der technischen Informatik ist das Verständnis und der Entwurf von digitalen Schaltungen. Dabei bilden *Schaltbilder* (*schematics*) das wesentliche Hilfsmittel zur Darstellung und sind damit gleichzeitig das wichtigste Beispiel für die im ersten Projektbericht eingeführte Kategorie der *Diagramm-Aufgaben*. Für die automatische Überprüfung von Übungsaufgaben aus dieser Kategorie von Aufgaben ist es also notwendig, Schaltbilder für Schaltnetze und Schaltwerke auf der Gatterebene und der Register-Transfer-Ebene zu repräsentieren und auszuwerten.

Simulation, Verifikation

In der industriellen Praxis des Schaltungsentwurfs werden derzeit vor allem zwei Techniken zur Überprüfung bzw. Verifikation eingesetzt, die einander perfekt ergänzen. Mittels *Simulation* lassen sich Modelle der Schaltung direkt erproben, während unter dem Begriff der *formalen Verifikation* alle Algorithmen zusammengefasst werden, mit denen sich Eigenschaften der Modelle mathematisch beweisen lassen. Eine dritte Gruppe von sogenannten *Test*-Verfahren dient dazu, gefertige Chips auf Fabrikationsfehler zu untersuchen. Ein aktueller Überblick der Thematik findet sich zum Beispiel in [13].

Softwareauswahl

Als Softwareplattform für die Übungsaufgaben zum Schaltungsentwurf ist das Simulations-Framework *Hades* vorgesehen [8], das am Fachbereich Informatik der Universität Hamburg entwickelt wird und von uns bereits seit einigen Jahren auch für Übungen, Tutorien und die Praktika eingesetzt wird [7, 6]. Die Software umfasst einen interaktiven Schaltplaneditor mit Hilfsmitteln zur Visualisierung und lässt sich einfach in die Matlab- bzw. Jython-basierten Skripte einbinden.

Die Entscheidung gegen den Einsatz kommerzieller Simulatoren beruht nicht nur auf deren komplexer Bedienung und den hohen Lizenzkosten, sondern vor allem auf dem Zeitaufwand zum Erlernen einer Hardwarebeschreibungssprache wie VHDL oder Verilog [13]. Im derzeitigen Informatik-Grundstudium in Hamburg steht weder ausreichend Zeit noch entsprechende Betreuungskapazität dafür zur Verfügung. Da praktisch alle kommerziellen Simulatoren über eine Skript-Schnittstelle gesteuert werden können, ist die nachträgliche Integration dieser Werkzeuge in die interaktiven Skripte bei Bedarf jederzeit möglich.

Zur automatischen Überprüfung einer Schaltung gibt es mehrere Möglichkeiten, wobei sowohl Simulation als auch Verifikation eingesetzt werden können:

- vollständige Simulation der Schaltung mit allen möglichen Eingabewerten und Vergleich mit einer Referenzschaltung.
- Simulation mit speziell vorbereiteten Eingabewerten (*stimuli*) und Vergleich mit den zugehörigen Referenzwerten.
- Simulation mit pseudozufälligen Eingabedaten und Signaturanalyse. Dieses Prinzip wird in Abschnitt 5.3 erläutert.
- Überprüfung von Eigenschaften der Schaltung, zum Beispiel Art und Anzahl der verwendeten Komponenten, nicht angeschlossene Eingänge, Kurzschlüsse, etc.
- Vergleich der Testschaltung mit einer Referenzschaltung über symbolischen Vergleich (*model-checking*).

Natürlich können auch mehrere dieser Verfahren kombiniert werden. Insbesondere das zuletzt genannte Prinzip des *model-checking* würde sich im Prinzip ideal zur Schaltungsüberprüfung eignen, da bei Abweichungen von der Referenzschaltung gleich die Eingabewerte bzw. Eingabefolgen protokolliert werden, unter denen der Fehler auftritt. Damit stellt model-checking eine ideale Ergänzung zu simulationsbasierten Werkzeugen wie Hades dar. Leider steht uns bisher aber keine ausreichend leistungsfähige Software dieser Art zur Verfügung, die sich einfach in die interaktiven Skripte integrieren ließe.

5.2 Interaktive Simulation

statische…

Die traditionelle Darstellung von Schaltnetzen und Schaltwerken als statische schwarz-weiße Diagramme ist zwar in gedruckten Lehrbüchern kaum zu vermeiden, gleichzeitig aber denkbar ungeeignet zur Vermittlung der meistens recht komplexen zeitabhängigen Funktion der Schaltungen. Während geübte Entwickler einem Schaltplan bereits auf den ersten Blick die grobe Funktion der Schaltung ansehen und dann die Details quasi „im Kopf simulieren“, kann diese Fähigkeit bei den Studierenden natürlich nicht vorausgesetzt werden. Sie muss vielmehr trainiert werden.

… vs. interaktive Schaltbilder

Ein wesentliches Hilfsmittel dazu ist die *interaktive Simulation* direkt während der Schaltungseingabe im Editor. Sofern die Software alle Datenstrukturen dynamisch verwaltet, werden Änderungen im Editor auch sofort in der Simulation sichtbar und die Schaltung kann jederzeit ausprobiert und getestet werden. Dies ist ein gewaltiger Vorteil gegenüber dem traditionellen Zyklus mit der Abfolge separater *Edit-Compile-Link-Simulate-Analyze*-Schritte, der weiterhin von den meisten kommerziellen Systemen eingesetzt wird, da sich auf diesem Weg die beste Simulationsperformance erreichen lässt. Angesichts der vergleichsweise kurzen Simulationszeiten für die Beispielschaltungen (jedenfalls im Grundstudium) ist der Performancegewinn für die Simulation jedoch vernachlässigbar, während das wiederholte Beenden, Compilieren und Neuaufsetzen der Simulation nach Fehlern ein echtes Problem darstellt. Demgegenüber lassen sich die gerade bei Anfängern unvermeidlichen Fehler bei einer interaktiven Simulation schnell beseitigen, ohne zeitaufwendige Compilerläufe abwarten zu müssen und die Simulation neu zu starten.

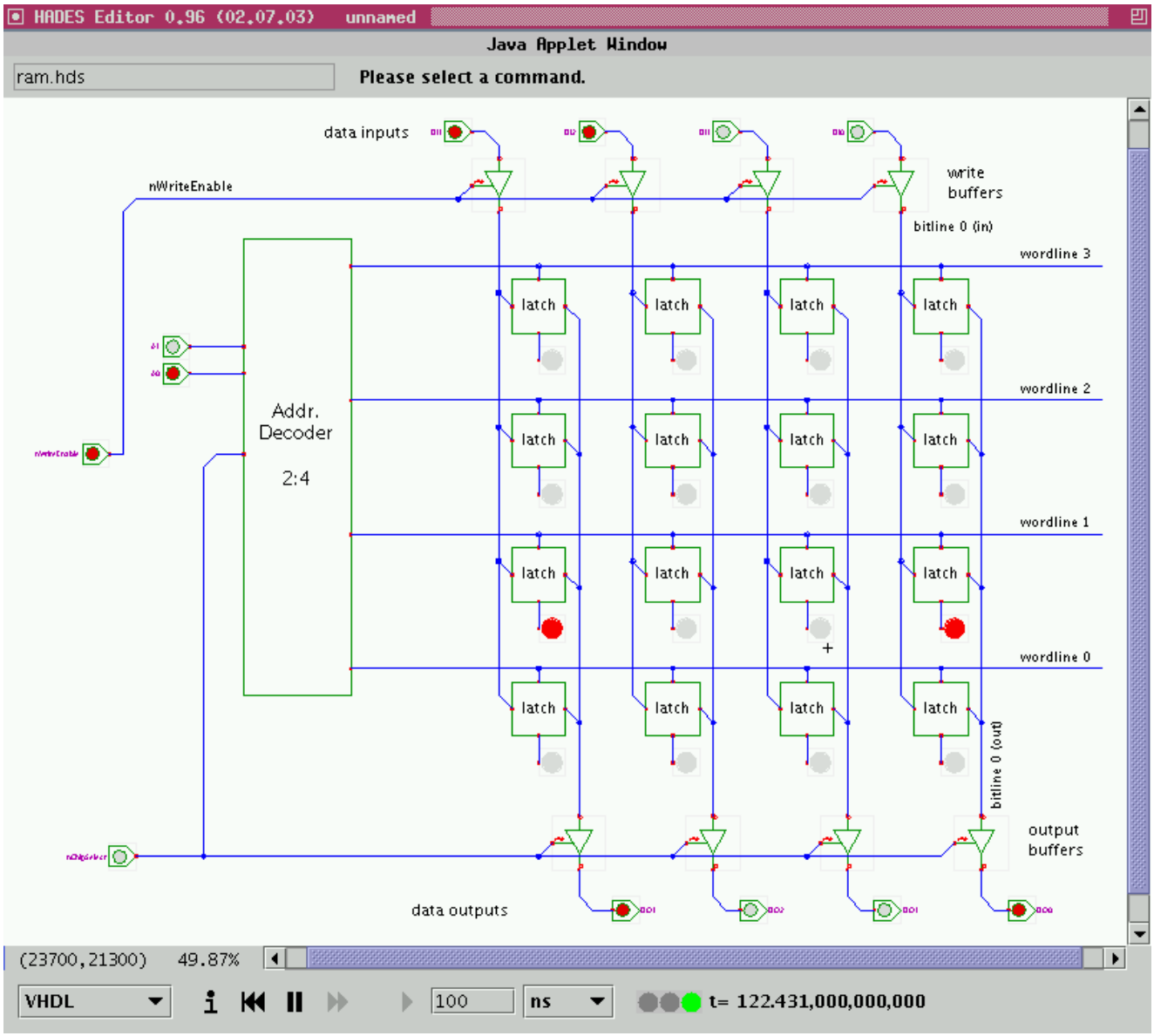


Abbildung 13: *Demonstration eines statischen RAM. Im interaktiven Modus kann die Funktionsweise des RAM durch Einstellen der Adress-, Daten- und Steuerleitungen direkt ausprobiert werden. Die Speicherinhalte aller einzelnen Zellen werden durch zusätzliche Leuchtdioden dargestellt.*

Die folgenden Abbildungen 13 und 14 zeigen zwei Screenshots des Hades-Editors mit im Rahmen des Projekts neu entwickelten Schaltungsbeispielen. Die erste Abbildung zeigt den Aufbau eines Schreib-Lese-Speichers (SRAM, *static random-access memory*) mit Adressdekoder, Speichermatrix und Ausgangstreibern. Die gesamte Schaltung kann interaktiv bedient werden, wobei die aktuellen Logikpegel auf allen Leitungen durch Farbkodierung visualisiert werden (*glow-mode*), so dass eine optimale Beobachtbarkeit der Schaltung sichergestellt ist. Die Visualisierung wurde allerdings für die Abbildungen in diesem Report teilweise abgeschaltet, um einen besseren Kontrast im Druckbild zu garantieren. Da die Schreib- und Lesevorgänge direkt ausprobiert werden können, ist ein Verständnis der Funktion wesentlich einfacher als bei der ansonsten notwendigen „Simulation im Kopf“ der Abläufe beim Betrachten einer statischen Abbildung in einem Lehrbuch wie [23].

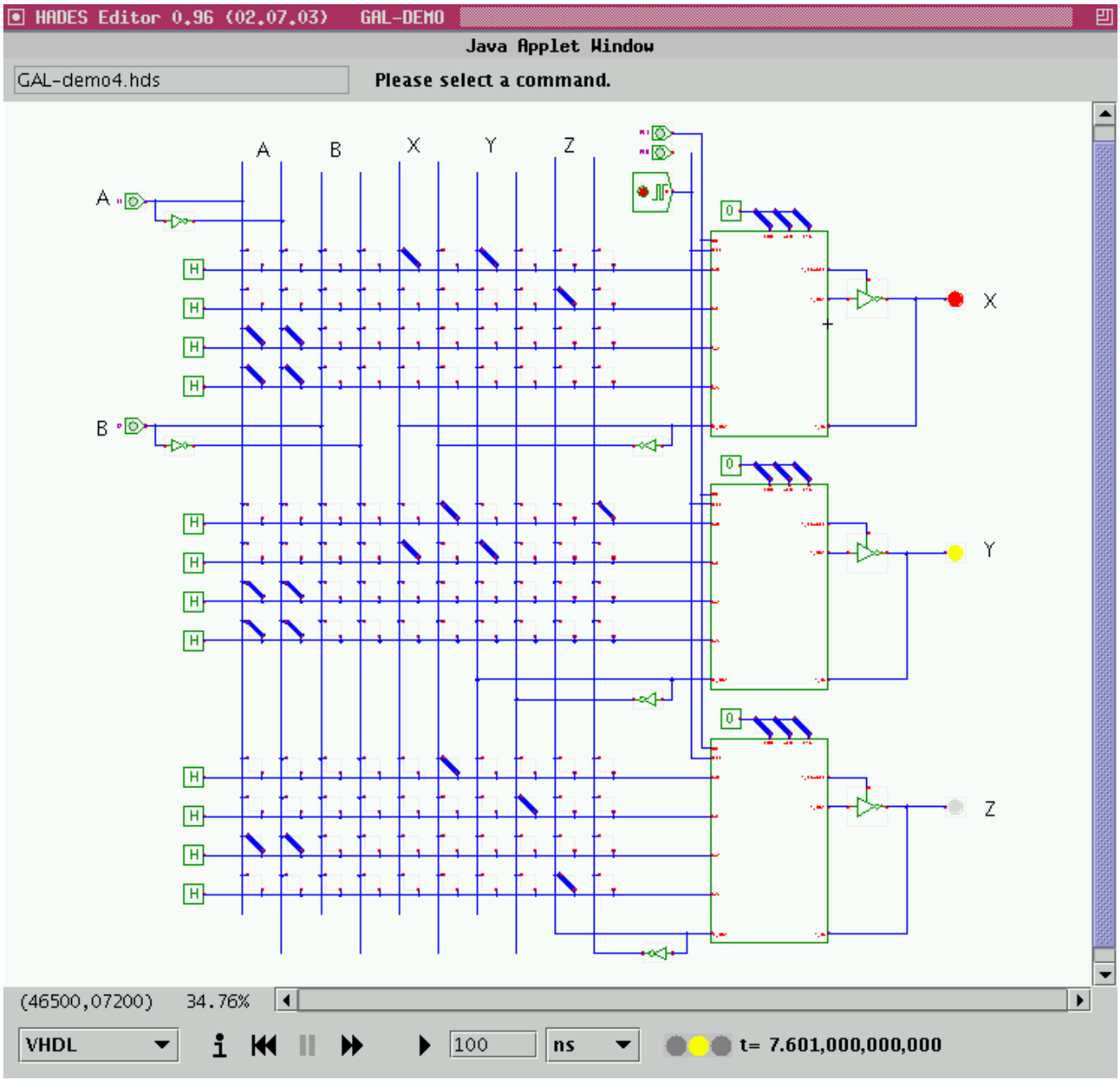


Abbildung 14: *Interaktive Simulation im Hades-Framework. Die Beispielschaltung demonstriert den typischen Aufbau eines PAL bzw. GAL-Bausteins (programmable/generic array logic). Der Zustand der programmierbaren Verbindungen zwischen den Eingangsleitungen (vertikal, A. . . Z) und Produkttermen (horizontal) wird durch die diagonalen Leitungen visualisiert. Die Programmierung kann durch einfaches Anklicken der entsprechenden Sicherung umgeschaltet werden. Mit der im Bild dargestellten Programmierung des GAL ergibt sich eine Ampelschaltung.*

Noch deutlicher wird dieser Vorteil in Abbildung 14, die den typischen Aufbau eines GALs zeigt. Diese programmierbaren Bausteine verwenden eine UND-ODER Architektur mit fest vorgegebenen ODER-Termen und vom Anwender über Sicherungen (*fuses*) in der UND-Matrix programmierbaren UND-Termen. Zusätzlich kann das Verhalten der Ausgangsblöcke programmiert werden, wobei ein ODER-Term direkt oder auf dem Umweg über ein D-Flipflop mit den Ausgängen und Rückkopplungsleitungen verbunden werden kann.

Da gängige Entwicklungsumgebungen für GALs (und FPGAs) nur sehr eingeschränkt über Interaktionsmöglichkeiten verfügen, ist die Simulation der Schaltungen mühsam. Die Programmierung der Bausteine setzt letztlich ein Verständnis des Funktionsprinzips und eine weitgehende Abstraktion bereits voraus. Die Hades-Schaltung dagegen ermöglicht es, die Sicherungen während der Simulation interaktiv einzustellen und die resultierende Funktion direkt zu beobachten. In der Abbildung werden intakte Sicherungen durch die diagonalen Verbindungen von den Eingabetermen (vertikal) zu den Produkttermen (horizontal) dargestellt, während offene Sicherungen aus dem Schaltbild entfernt werden.

MatlabAdapter

Mit den im Rahmen des Projekts entwickelten Hilfsklassen lässt sich das Hades-Framework problemlos von Matlab aus ansteuern und kann damit nahtlos in die bestehenden interaktiven Skripte integriert werden. Als Beispiel ruft das folgende Matlab-Skript den Hades-Editor auf, lädt die Beispielschaltung aus Abbildung 14 und schaltet nach einigen Sekunden Simulation den Eingang A um:

```
adapter = hades.utils.MatlabAdapter;   % Matlab-Interface
adapter.setMatlabPath( path );         % initialisieren
adapter.loadProperties( 'hades.ini' ); % Voreinstellungen

editor = hades.gui.Editor;             % Editor öffnen
editor.getFrame.setSize( 1024, 700 );  % Fenstergröße
editor.doOpenDesign( 'GAL.hds', 0 );   % Schaltung laden
editor.getSimulator.runForever;        % Simulation starten

switchA = editor.getDesign.getComponent( 'A' )
switchA.assign( '1',  3.0 );           % nach  3 Sekunden
switchA.assign( '0', 10.0 );           % nach 10 Sekunden
```

Wie das Beispiel zeigt, können von Matlab aus direkt Simulationsereignisse erzeugt und an den Simulator übergeben werden. Zusammen mit dem Zugriff auf die aktuelle Schaltung („design“) und den Simulationskern („simulator“) können also auch Animationen vorbereitet und abgespielt werden.

5.3 Schaltungsüberprüfung mittels Built-In Selftest

vollständige Simulation,

Zur Überprüfung einer digitalen Schaltung mittels Simulation gibt es die bereits in Abschnitt 5.1 skizzierten Möglichkeiten, die sich durch die Auswahl der Eingabedaten für die Simulation unterscheiden. Bei einer *vollständigen* Simulation wird das Verhalten der Schaltung für alle überhaupt möglichen Eingabedaten berechnet und mit den erwarteten Werten verglichen. Wegen der mit der Anzahl der Eingänge und internen Zustände exponentiell wachsenden Anzahl der möglichen Werte kommt dieses Verfahren jedoch nur für Schaltnetze mit wenigen Eingängen in Frage und verbietet sich für komplexe Schaltnetze oder Schaltwerke. Deshalb ist es notwendig, die Menge der Eingabedaten für die Simulation drastisch zu reduzieren. Häufig ist es möglich, spezielle Folgen von Eingabedaten (*stimuli*) vorzubereiten, unter denen alle Zustände der Schaltung durchlaufen werden und gezielt alle Übergänge und Ausgangswerte überprüft werden können. Der Aufwand zur Erstellung solcher Eingabedaten ist jedoch sehr hoch.

ausgewählte Stimuli,

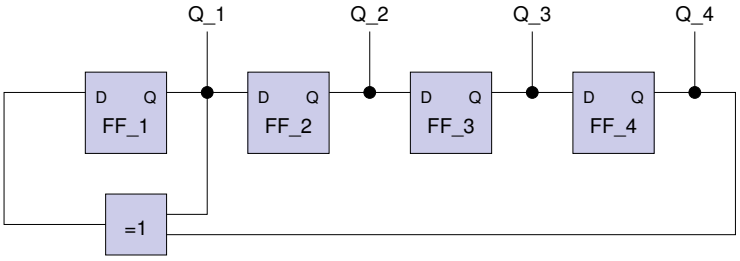


Abbildung 15: *Aufbau eines 4-bit rückgekoppelten Schieberegisters (LFSR, linear-feedback shift-register). Bei geeigneter Wahl der XOR-Gattern im Rückkopplungszweig durchläuft ein n-bit Register alle* $2^n - 1$ *Zustände außer* $0 \ldots 0$ *in pseudozufälliger Reihenfolge.*

oder pseudo-zufällige Eingabedaten

Als dritte Alternative ist es möglich, zufällige bzw. pseudozufällige Eingabemuster für die Simulation zu verwenden, die sich sehr einfach und effizient erzeugen lassen. Dieses Prinzip wird häufig mit einer Signaturanalyse der Ausgangswerte anstelle der detaillierten Überprüfung aller Ausgangswerte kombiniert. Da bei zufälligen Eingabedaten natürlich nicht mehr sichergestellt werden kann, dass wirklich alle Zustände der zu testenden Schaltung erreicht und überprüft werden, lassen sich mit dieser Methode nicht alle Fehler in einer Schaltung aufspüren. Dieser prinzipielle Nachteil kann jedoch in der Praxis häufig durch eine entsprechend höhere Anzahl der Eingabedaten kompensiert werden, um die Wahrscheinlichkeit übersehener Fehlerquellen auf ein akzeptables Maß zu reduzieren.

LFSR-Prinzip

Eine spezielle Ausprägung dieser Verfahren basiert auf dem Einsatz von linear-rückgekoppelten Schieberegistern. Der Aufbau eines solchen LFSR (*linear-feedback shift-register*) ist in Abbildung 15 skizziert. Dabei werden n Flipflops als Schieberegister hintereinander geschaltet und die Ausgänge mehrerer Flipflops über einen Baum von XOR-Gattern auf den Eingang des ersten Flipflops rückgekoppelt. Abhängig von der Auswahl der XOR-Gatter im Rückkopplungszweig durchläuft ein derartiges Register eine eindeutige Folge von Zuständen, die auf den ersten Blick jedoch zufällig erscheint und deshalb als *pseudozufällig* bezeichnet wird. In bestimmten Fällen umfasst diese Folge alle möglichen $2^n - 1$ Zustände außer der Null. Als Erweiterung ist es möglich, externe Eingangssignale über weitere XOR-Gatter in das Register einzuspeisen. Abhängig von den Eingangsdaten verändert sich dann auch die Folge der Zustände, die das Register durchläuft. Eine Einführung in die Theorie der LFSR-Register findet sich zum Beispiel in [27].

Signaturanalyse

Der Zustand des Registers nach einer Anzahl von Takten kann als digitale Signatur interpretiert werden. Falls die an einer konkreten Schaltung beobachtete Signatur von der theoretisch erwarteten Signatur abweicht, weist die beobachtete Schaltung mit Sicherheit einen Fehler auf. Andererseits kann es vorkommen, dass eine defekte Schaltung zufällig die korrekte Signatur liefert. Die Wahrscheinlichkeit dafür kann abhängig von der Anzahl n der Flipflops in den LFSR-Registern zu $p \approx 2^{-n}$ abgeschätzt werden. Für einen typischen Wert wie $n = 32$ ergibt sich bereits eine extrem kleine Wahrscheinlichkeit von $p \approx 2.3 \cdot 10^{-10}$. Wegen des geringen Hardwareaufwands werden derartige Schieberegister häufig direkt in VLSI-Mikrochips

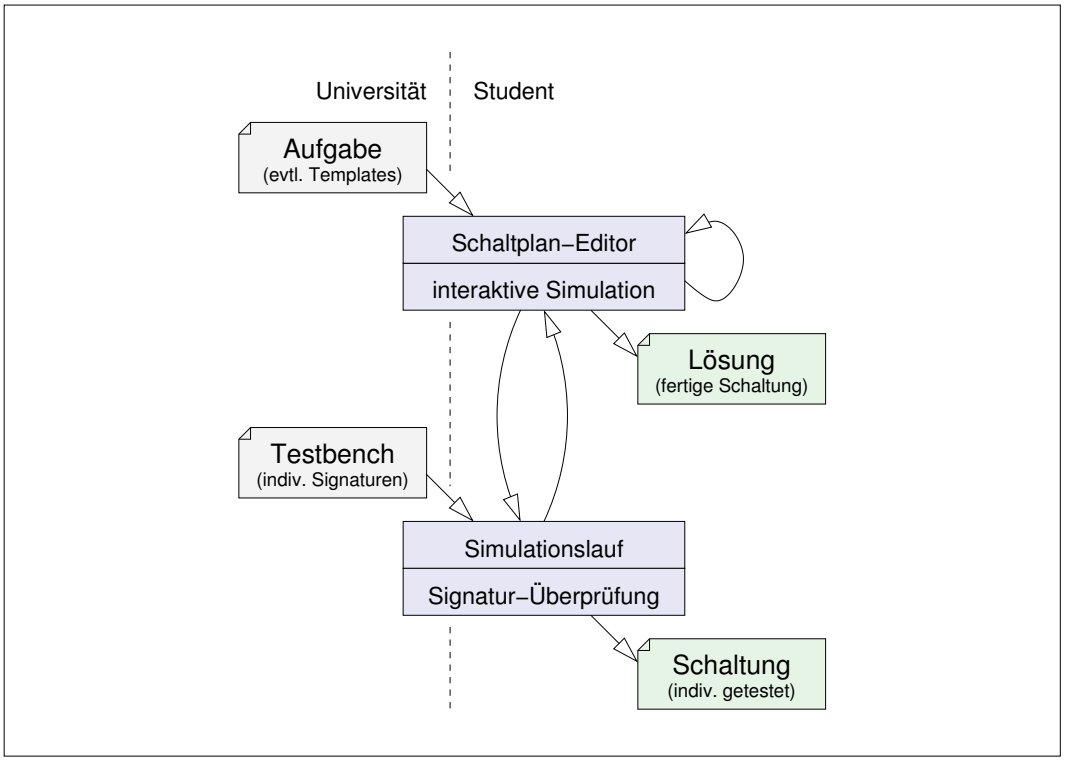


Abbildung 16: *Flussdiagramm zum Entwurf und zur Überprüfung digitaler Schaltungen mittels Signaturanalyse. Die Funktion der Schaltung kann bereits während der Schaltungseingabe interaktiv erprobt werden. Aus der anschließenden vollautomatischen Simulation der Testbench ergibt sich eine digitale Signatur, die nur bei korrekter Funktion der Schaltung den erwarteten Wert annimmt. Zusätzliche Plausibilitäts-Checks erlauben es, über die Überprüfung hinaus auch Hilfestellungen zu erzeugen.*

integriert und zum Selbsttest der Schaltungen bei voller Taktfrequenz eingesetzt (*built-in selftest*, BIST).

Zur automatischen Überprüfung einer Schaltung nach diesem Prinzip wird eine kleine Testumgebung (*testbench*) benötigt, die lediglich die folgenden Komponenten enthält:

- die zu testende Schaltung selbst, die in der Literatur üblicherweise als UUT (*unit under test*) bezeichnet wird,
- ein erstes LFSR mit geeignet ausgewählter Anzahl von Stufen als Generator für die pseudozufälligen Eingangswerte der UUT,
- ein zweites, als Signaturanalyse-Register geschaltetes LFSR zur Berechnung der digitalen Signatur der Ausgangswerte der UUT,
- weitere Hilfskomponenten wie Taktgenerator und Resetgenerator.

Zur Vorbereitung der Simulation werden zunächst die Startzustände des Generators und des Signaturanalyseregisters einmal auf geeignete Werte gesetzt. Danach wird die Simulation gestartet und für eine vorher festgelegte Anzahl von Takten ausgeführt. Nach Abschluß der Simulation kann der endgültige Signaturwert aus dem Analyseregister ausgelesen und mit dem bekannten Wert einer korrekten Schaltung verglichen werden.

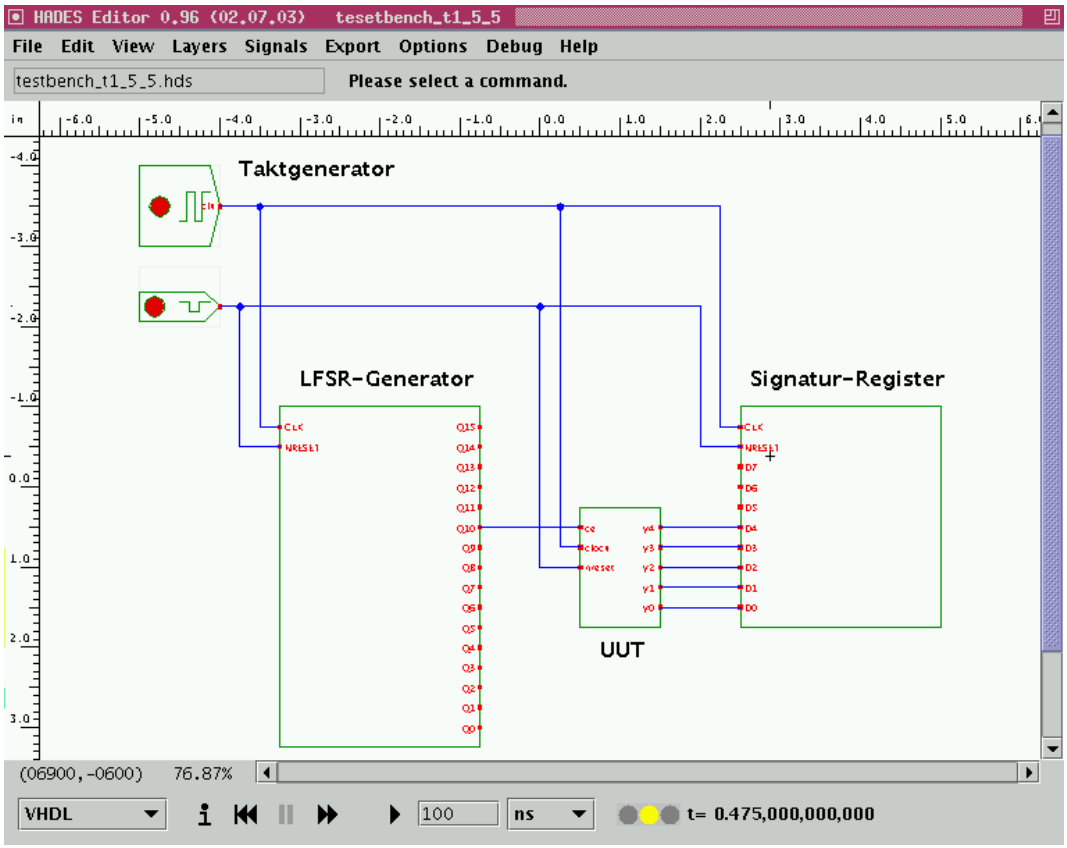


Abbildung 17: *Typischer Aufbau einer Testumgebung (*testbench*). Ein erstes LFSR-Register erzeugt pseudozufällige Eingabedaten für die zu testende Schaltung UUT (*unit-under-test*), während ein zweites Register aus den Ausgaben der Schaltung die zugehörige digitale Signatur berechnet. Nach Simulation einer vorher festgelegten Anzahl von Testmustern wird die Signatur ausgelesen und mit dem bekannten Wert einer korrekten Schaltung verglichen. Die Simulation kann entweder (wie in der Abbildung) im interaktiven Modus mit Benutzeroberfläche oder auch im reinen Textmodus durchgeführt werden.*

Einbettung in die Skripte

Für die Übungsaufgaben zum Schaltungsentwurf wird dieses Vorgehen mit der interaktiven Simulation kombiniert. Der gesamte Ablauf ist in Abbildung 16 skizziert. Zusätzlich zu den Texten der Übungsaufgaben werden den Studierenden noch zwei Hades-Schaltungen zur Verfügung gestellt: die Testbench mit den LFSR-Komponenten und eine leere Template-Schaltung, die bereits alle Ein- und Ausgabekomponenten der zu entwickelnden Schaltung enthält, womit eine fehlerfreie Kommunikation zwischen Testbench und UUT garantiert werden kann. Die Aufgabe der Studierenden ist es jetzt, die Template-Schaltung zu vervollständigen, um die geforderte Funktion zu realisieren. Die Schaltung kann dabei im Hades-Editor jederzeit mit Hilfe der interaktiven Simulation überprüft werden.

Sobald die Schaltung im Rahmen der interaktiven Tests korrekt zu funktionieren scheint, ist es Zeit, die Schaltung abzuspeichern. Anschließend wird die Testbench in den Simulator geladen und der oben skizzierte Ablauf gestartet, um die digitale Signatur zu berechnen. Aus einer Abweichung von der vorgegebenen Signatur der Musterlösung folgt dabei sofort, dass noch ein Problem in der Schaltung vorliegt. Bei Übereinstimmung der Signaturen ist die Schaltung dagegen mit sehr hoher Wahrscheinlichkeit korrekt, so dass die Schaltung als Lösung eingeschickt

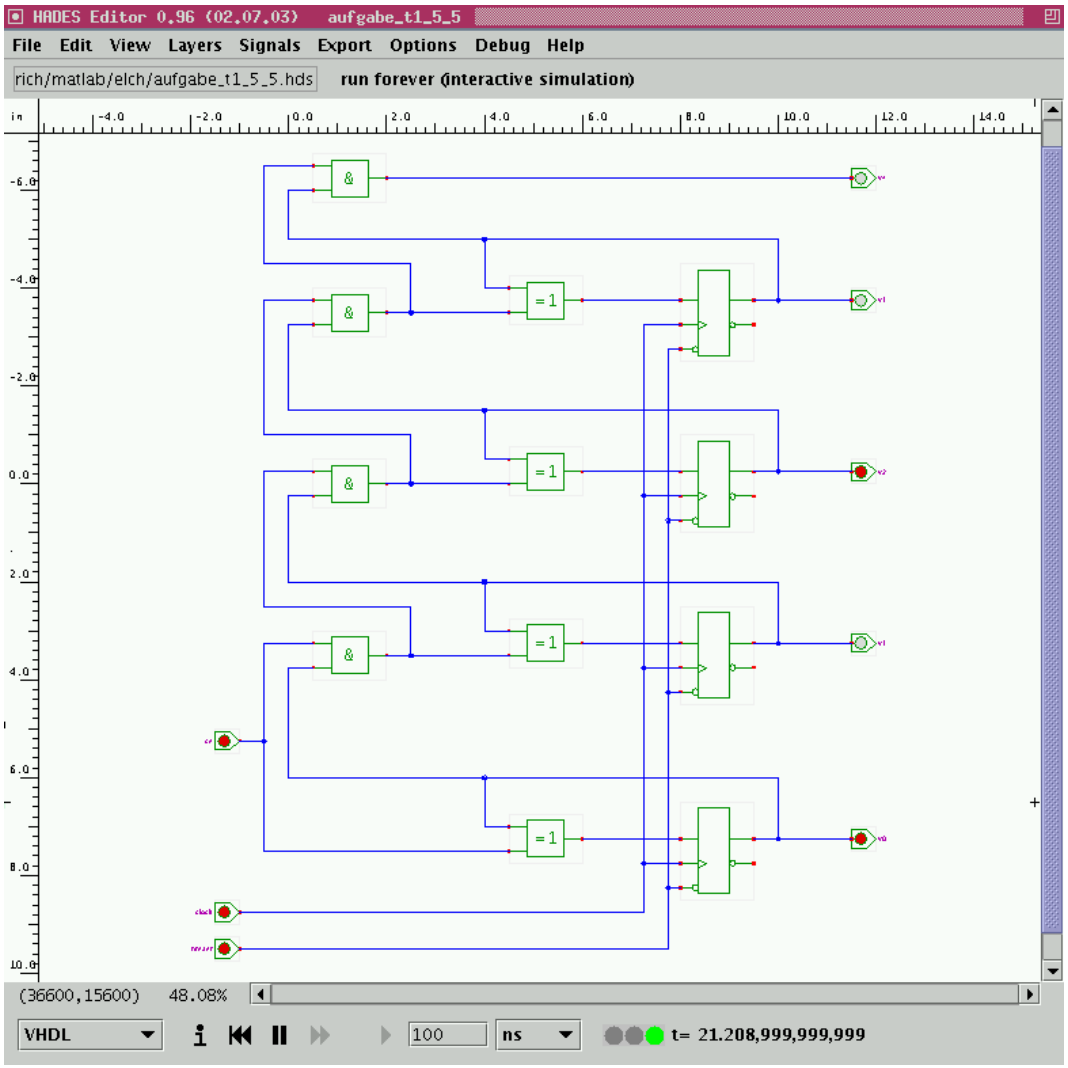


Abbildung 18: *Synchroner 4-bit Zähler (Aufgabe T1.5.5) als Beispiel zum Entwurf digitaler Schaltungen mit dem Hades-Framework.*

werden kann. Durch unterschiedlich gewählte Startwerte der LFSR-Komponenten ist es möglich, personalisierte Tests durchzuführen.

Ein konkretes Beispiel für eine solche Testumgebung als Hades-Schaltbild zeigt Abbildung 17. Bei der zu testenden Schaltung handelt es sich um einen einfachen synchronen 4-bit Zähler mit Enable-Eingang. Eine zugehörige Musterlösung mit vier D-Flipflops und dem typischen Aufbau der Zählerlogik ist in Abbildung 18 dargestellt. Da die zu testende Schaltung nur einen Eingang und nur vier Ausgänge umfasst, werden die überzähligen Anschlüsse der LFSR-Komponenten einfach freigelassen.

Performance

Die Anzahl der notwendigen Eingangsmuster hängt stark von den Eigenschaften und der Komplexität der zu überprüfenden Schaltung sowie der geforderten Entdeckungswahrscheinlichkeit von Fehlern ab. Während im Bereich des Hardware-Selbsttests mehrere Millionen Testmuster kein Problem darstellen, da die Überprüfung mit der vollen Taktfrequenz des Systems betrieben wird, gilt dies für einen Softwaresimulator wie Hades selbstverständlich nicht. Wegen der verhältnismäßig geringen Komplexität der in den Übungsaufgaben zur technischen Informatik vorkommenden Schaltungen dürften in den meisten Fällen aber einige Zehntausend

Testmuster ausreichen. Dies entspricht auf aktuellen Rechnern Simulationslaufzeiten im Bereich von wenigen Sekunden, so dass der Selbsttest problemlos auch mehrfach ausgeführt werden kann.

Als konkretes Beispiel für die Umsetzung dieser Funktionen in die Software-Umgebung des interaktiven Skripts zeigt Abbildung 19 eine Matlab-Funktion, die die oben geschilderten Schritte automatisch ausführt. Für die automatische Überprüfung ist es natürlich nicht notwendig, die graphische Oberfläche des Hades-Systems zu starten. Statt dessen wird lediglich die angegeben Testbench-Schaltung in den Simulator geladen. Anschließend werden die Startwerte (*seed*) der LFSR-Register eingestellt und die Simulation für die angegebene Anzahl von Takten gestartet. Die Funktion liefert als Rückgabewert den am Ende der Simulation vorliegenden Signaturwert an das aufrufende Skript zurück. *bist.m*

Offensichtlich ist es nicht möglich, aus einer gegebenen Signatur die Schaltung zu rekonstruieren, die während der Simulation gerade diesen Signaturwert erzeugt. Es ist daher anders als bei den in in Abschnitt 4.3 beschriebenen Aufgabentypen überhaupt kein Problem, wenn die erwarteten Werte der Signaturanalyse direkt im Klartext in den Skripten eingetragen sind.

Anstelle der pseudozufälligen Eingabedaten können natürlich auch speziell für eine Aufgabe vorbereitete Stimuli für die Simulation der Testbench verwendet werden. Die kürzeren Simulationszeiten und die verbesserte Überprüfung bestimmter Teilfunktionen, die von den pseudozufälligen Eingabedaten eventuell nicht überprüft werden, werden hierbei mit dem höheren Aufwand zur Erstellung der Eingabedaten erkauft.

Erst der praktische Einsatz der Software wird zeigen, ob es bei Bearbeitung der Aufgaben in großem Umfang zum „Abschreiben“ kommt. Offenbar ist es möglich, dass die Studierenden ihre Schaltungen untereinander austauschen, bevor sie die Testbench simulieren. Daher müssen eventuell zusätzliche strukturelle Eigenschaften der Schaltungen ausgewertet werden, um Übereinstimmungen zu entdecken. Beispiele solcher Merkmale sind die Anzahl, Art, Position und Namen der einzelnen Komponenten und Signale in der Schaltung. Ebenfalls denkbar wäre die Verwendung verschlüsselter oder digital signierter Design-Dateien, die sich nur nach Eingabe eines persönlichen Codes öffnen und bearbeiten lassen.

5.4 Hilfestellungen

Das im vorigen Abschnitt beschriebene Verfahren reicht aus, um mit geringem Aufwand eine digitale Schaltung zu überprüfen. Leider liefert die Signaturanalyse aber keinerlei Hinweis auf die eigentliche Fehlerursache, falls die berechnete von der erwarteten Signatur abweicht. Für brauchbare Hilfestellungen muss die Signaturanalyse also um weitere Tests ergänzt werden. Dabei werden, abhängig von der konkreten Übungsaufgabe, eine oder mehrere der folgenden Techniken zum Einsatz kommen:

- Kombination von Signaturanalyse mit geeignet vorbereiten Eingabedaten und (vorher aufgezeichneten) Werteverläufen der erwarteten Ausgangsdaten. Dies

```
function signature = bist( testbenchname, seed, n_cycles )
%
% bist: run a HADES simulation on the specified testbench.
% The function returns the final signature value of the
% analyzer register.

% load the testbench design
%
manager   = hades.manager.DesignManager.getDesignManager;
design    = manager.getDesign(  [], testbenchname, 0 );
simulator = hades.simulator.VhdlBatchSimKernel;
design.setSimulator( simulator );
simulator.setDesign( design );

% reference the components and specify some options
%
generator = design.getComponent( 'generator' );
analyzer  = design.getComponent( 'analyzer' );
clockgen  = design.getComponent( 'clockgen' );
generator.setSeed( seed );
analyzer.setSeed( seed  );
clockgen.setAutoStopCycles ( n_cycles );
clockgen.setPeriod( 1.0E-3 );

% now start the simulation
%
maxSimTime = 20.0;
simulator.runFor( maxSimTime );

% wait in this thread until the simulation is finished; we
% break after 100 iterations of 0.5 seconds as a fallback
%
for k=1:100
  pause( 0.5 );
  if (simulator.getSimTime >= maxSimTime)
    break
  end
end

simulator.stopSimulation;   % force a stop
tmp = analyzer.getValue;
signature = java.lang.Integer.toString( tmp );
```

Abbildung 19: *Matlab-Funktion* bist.m *zur automatischen Überprüfung einer Hades-Testumgebung. Nach Erreichen der vorgegebenen Anzahl von Taktzyklen wird der Signaturwert aus dem Analyse-Register ausgelesen und als Resultatwert zurückgegeben.*

erlaubt eine detaillierte Aussage darüber, zu welchem Zeitpunkt der erste Fehler auftritt, oder für welche Eingabedaten die Ausgangswerte noch Fehler aufweisen.

- Bei Schaltungen mit mehreren Ausgängen lässt sich eine Signaturanalyse *getrennt für alle einzelnen Ausgänge* durchgeführen. Abhängig von der Aufgabenstellung ist eventuell auch die Analyse interner Signale der Schaltung möglich (sofern diese eindeutig zugeordnet werden können). Während die im letzten Abschnitt beschriebene globale Signaturanalyse bereits bei Vorliegen eines einzelnen kleinen Fehlers die pauschale Antwort „falsch“ für die gesamte Schaltung liefert, erlaubt diese separate Analyse immerhin eine Unterscheidung zwischen den einzelnen Ausgängen. Dies sollte sich auch positiv auf die Motivation der Studierenden auswirken, da sich sukzessive Verbesserungen der Schaltung entsprechend bereits in den Zwischenergebnissen niederschlagen.

- Unabhängig von den Anforderungen der jeweiligen Aufgaben lassen sich jederzeit *strukturelle Merkmale* der gerade im Editor aktiven Schaltung auswerten und auf Plausibilität überprüfen. Zum Beispiel kann für alle in der Schaltung vorhandenen Komponenten überprüft werden, ob alle Eingänge korrekt beschaltet sind oder ob Kurzschlüsse vorliegen. Weitere wichtige Tests betreffen die Takt- und Resetanschlüsse aller vorhandenen Flipflops, da Fehler an diesen Stellen besonders kritisch sind. Viele dieser Funktionen sind bereits im Hades Editor integriert und können von den interaktiven Skripten heraus direkt aufgerufen werden.

- Auch wenn bei einigen Übungsaufgaben mehrere korrekte Lösungen oder gleichwertige Varianten existieren, lassen sich meistens zusätzliche Bedingungen ableiten, deren Überprüfung weitere Hilfestellungen für die Studierenden liefern kann. Zum Beispiel lässt sich beim Entwurf von Schaltnetzen häufig die *Mindestanzahl* der benötigten Gatter berechnen oder bei Schaltwerken die Mindestanzahl der Flipflops. Entsprechende Werte können vom Skript heraus vorgegeben und dann mit der realisierten Lösung verglichen werden.

- Bei einigen Übungsaufgaben kommt es auf die Details des *Zeitverhaltens* der Schaltung an. Typische Beispiele liefern Aufgaben zum Themengebiet Hazards oder die verschiedenen Realisierungsvarianten von Addierern mit ihrem Spektrum von Laufzeiten und Hardwareaufwand. Da der Hades-Simulator Gatterlaufzeiten simuliert, ist es durch geeignete Voreinstellung der Gatterverzögerungszeiten und des Taktes der Testumgebungen möglich, auch Laufzeiteffekte in die Überprüfung mit einzubeziehen.

- Neben den oben skizzierten *statischen* lassen sich während der Simulation auch *dynamische* Merkmale der Schaltung auswerten. Ein gebräuchlicher Test ist ein Protokoll darüber, ob alle Signale in der Schaltung nach dem Reset einen definierten Wert annehmen und während der Simulation mindestens einmal ihren Wert wechseln.

  Bei Bedarf sind auch aufwendigere Checks dieser Art möglich. Ein Beispiel ist die Überprüfung, ob in einem Mikroprogramm die Write-Enable-Signale

überhaupt und/oder in der richtigen Reihenfolge aktiviert werden. Erfahrungen mit dem Praktikum Technische Informatik zeigen, dass die Studierenden schon durch einfache Hinweise auf solche Problemursachen viel Zeit beim Bearbeiten der Aufgaben sparen können.

- Schließlich stellt das Hades-Framework auch die übliche *Impulsdiagramm*-Darstellung der Signalverläufe während der Simulation zur Verfügung, allerdings ohne besondere Softwareunterstützung zur Auswertung der Impulsdiagramme. Zum Erkennen fehlerhafter Werte und insbesondere für das Zurückverfolgen zur Fehlerursache sind entsprechende Vorkenntnisse erforderlich. Dies ist aber für die automatische Überprüfung kein Problem. Durch Aufzeichnung und Auswertung der Impulsdiagramme ist zum Beispiel der oben genannte Test der Umschalthäufigkeiten der einzelnen Signale problemlos möglich. Nicht schaltende Signale lassen sich dann leicht ermitteln und etwa im Editor hervorheben.

Zum Schluss sei noch einmal darauf hingewiesen, dass die interaktive Softwareumgebung des Hades-Frameworks selbst bereits umfangreiche Hilfestellungen anbietet. Der zentrale Vorteil gegenüber dem Schaltungsentwurf mit Papier und Bleistift ergibt sich bereits aus der Möglichkeit, die aktuell entworfene Schaltung sofort auszuprobieren. Aber auch gegenüber realen Hardwareprototypen auf Experimentierplatinen ergeben sich Pluspunkte, etwa durch die leichtere Editierbarkeit inklusive Undo-Optionen. Nicht zuletzt ist die Beobachtbarkeit der Schaltungen durch die Farbkodierung der Signalpegel optimal, so dass problematische Bereiche einer Schaltung buchstäblich auf einen Blick zu erkennen sind.

6 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 derzeit vor allem Matlab. Das Prinzip lässt sich aber mit jeder Programmierumgebung umsetzen, die das interaktive Ausführen von Programmtexten oder Skripten erlaubt. Gerade das bereits als Prototyp implementierte Konzept der Einbettung von kleinen Java-Applets in interaktive HTML-Seiten dürfte in Zukunft eine interessante Alternative darstellen, da alle Funktionen der bekannten Web-Browser zur Verfügung stehen und die Anwender ihre gewohnte Umgebung vorfinden.

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 drei der sechs im ersten Bericht [10] aufgestellten Klassen von Aufgaben vorgestellt: Erstens Auswahlaufgaben inklusive Multiple-Choice, zweitens Zahlenwert-Aufgaben und drittens die Überprüfung von digitalen Schaltungen. Entsprechende Algorithmen für die Überprüfung von Formeln befinden sich in Vorbereitung und werden in Kürze zur Verfügung stehen. 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, D. R. O'Hallaron, *Computer Systems, A Programmer's Perspective*, Prentice Hall, 2003, ISBN 0-13-034074-X

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

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

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

[5] 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/

[6] 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/

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

[8] 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[28] 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

[29] 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

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