Dokumentation zu DigSim
Projekt in Programmierung I
Universität Fribourg SH'04/05

Urs Sieber
Bahnstrasse 9a
CH-9323 Steinach

Steinach, den 26. Mai 2005

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die Einleitung soll einen qualitativen Überblick über das Projekt geben. Die Dokumentation der Rahmenbedingungen zum Projekt wie auch die einzelnen Projektierungsphasen werden kurz angeschnitten und erläutert. Zum Schluss wird eine kurze Übersicht über die Gliederung dieser Dokumentation einen vereinfachten Zugang zur Beschreibung des Projekts schaffen.

1.1 Zweck des Projektes, Beschreibung der Aufgabe

Digitale Schaltkreise ermöglichten einst die Entwicklung von Computern und sind aus dem heutigen, technisierten Leben nicht mehr wegzudenken. Viele Alltagsgegenstände sind aus elementaren digitalen Einheiten aufgebaut und erleichtern uns heute das Leben. Dass im Innern dieser Gegenstände eine Mikrowelt von Einsen und Nullen agiert und steuert, bleibt den Anwendern meist verborgen. Man darf ohne Übertreibung behaupten, dass digitale Schaltkreise unser Leben mitgestalten, wenn nicht sogar leiten.

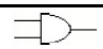
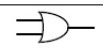
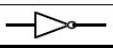
Gate	Symbol	Schreibweise
AND		$A \wedge B$
OR		$A \vee B$
NOT		$\overline{A}$

Abbildung 1: Primitive Bauteile digitaler Schaltkreise

Zweck des Projektes ist es, einen Simulator für digitale Schaltkreise in der eigenen Sprache DigSim zu entwickeln. Eine abstrakte Form zur Implementierung eines Schaltplanes mit den drei primitiven Bauteilen (Abbildung 1) Inverter (not-gate), and-gate und or-gate soll ermöglichen, den Ablauf der Signalleitung bis zum Output zu verfolgen und so ein Resultat herauszugeben. Weiter soll der Prozess der Signalverarbeitung in den einzelnen primitiven Bauteilen auch zeitabhängig sein. Verschiedene Bauteile benötigen unterschiedliche Zeiteinheiten zur Verarbeitung der Inputs. Die spezifischen Zeiteinheiten der Bauteile bestimmen also die Zeit, die vergeht, bis die Eingaben in den digitalen Schaltkreis als verarbeitetes Resultat ausgegeben werden können. Zusammenfassend erreicht man nach der abstrahierten Eingabe des Schaltkreises und Festlegung der einzelnen Zeitverzögerungen in den primitiven Bauteilen ein Ausgabesignal und die zu dessen Verarbeitung verstrichene Zeit .
Ein metazirkulärer Evaluator, das heisst, ein Evaluator, der in derselben

Sprache geschrieben ist, in der er auswertet (nach Abelson 1996 [2], p. 378), bearbeitet die Eingaben von DigSim. Basierend auf dem Lisp-Dialekt Scheme wird ein Evaluator, der ebenfalls als Lisp-Programm implementiert wird, programmiert. Die Auswertung stützt sich auf ein zu implementierendes Umgebungsmodell, das Prozeduranwendungen, elementare Prozeduren, aber auch beispielsweise Zuweisungen hinreichend beschreiben kann.
Schliesslich sollen die genannten Funktionen mit Hilfe von MrEd (Erweiterung von Scheme) mit einer grafischen Benutzeroberfläche versehen und das gesamte Projekt in eine stand-alone Applikation verpackt werden.
Ein wichtiger Aspekt der Arbeit soll ein klar und übersichtlich geführtes Benutzerhandbuch darstellen. In diesem sind Programm, Anwendung und Rahmendaten verständlich aufgeführt und dokumentiert.

Allgemein gilt das Augenmerk nicht nur dem endgültigen finalen Programm, sondern wird ebenfalls auf die verschiedenen Entwicklungsstufen und Nacharbeiten (Dokumentation) gerichtet.

1.2 Projektablauf, zeitlicher Rahmen

Während 15 Arbeitswochen mit Beginn im Oktober 2004, wird jeweils in einer Projekt-Vorlesung ein neuer Aspekt des Projekts besprochen und durch die Betreuer eingeführt. Mit Übungen zu diesen Vorlesungen soll der neue Stoff erarbeitet werden.

Die ersten vier Wochen sind einer Einführung in digitale Schaltkreise gewidmet. Ebenfalls wird das Ergebnis, die Sprache DigSim, bereits erläutert. Es folgen weitere vier Wochen Einführung in die Erarbeitung grafischer Steuerelemente und GUI's mit Scheme und MrEd. Die restlichen Wochen werden für die Implementierungen des Evaluators und des Umgebungsmodells verwendet. Zwei bis drei Wochen sind am Schluss noch für die Kompilation und Dokumentation aufzuwenden.
Mit zwei zu erreichenden Meilensteinen werden die Schranken fürs Projekt durch die Betreuung gesetzt. Meilenstein #1, 09.01.2005, beinhaltet die Grundmauern der grafischen Oberfläche und die Erzeugung einer standalone Applikation. Meilenstein #2, März 2005, gibt die Implementierung der DigSim-Erweiterungen in den Evaluator und Einbettung dieses in den Meilenstein #1 vor. Es bleibt so genügend Zeit übrig für die Redaktion des Benutzerhandbuches und für ausführliche Tests mit dem Projekt bis zum Abgabetermin am **31. Mai 2005**.

1.3 Verwendete Hard- und Software

Als Programmierungsoberfläche wurde PLT Scheme / DrScheme verwendet, das auf einem PC mit Windows XP Betriebssystem ausgeführt wurde. Die einfache Installation von Scheme auf einem Rechner erleichterte eine Bearbeitung des Projektes auf diversen unterschiedlichen Computern (Universität Fribourg, zu Hause, bei Freunden). Es können keine definierten Hardwareangaben gemacht werden, zumal von mehreren unterschiedlichen Systemen aus gearbeitet wurde. Es wurde aber ausschliesslich auf Windows-Systemen gearbeitet.

Die Projektdokumentation wurde mit Latex redigiert und im PDF-Format ausgegeben. Windows Editor WinEdt 5 diente als Editor für den Text der Dokumentation. Zur Kompilierung der Tex-Dateien wurde MikTex mit der Total-Installation gewählt und verwendet. Für die Bearbeitung der Screenshots und der Bilder wurde Adobe Photoshop 7.0 verwendet.

1.4 Gliederung der Projektdokumentation

In den folgenden Kapiteln dieser Dokumentation werden Projekt und dessen Rahmen näher erläutert:
In Kapitel 3 wird Anwendung und Benutzung der Applikation erklärt. Ein Beispiel in 3.4 gibt eine vertiefte und nahe Einsicht in die Benutzerregeln. Kapitel 4 beschränkt sich auf die Programmierung der Applikation. Ordnung der einzelnen Source-Files, Erklärung der Implementierung von Evaluator und Umgebungsmodell sind wichtige Bestandteile dieses Kapitels. Weiter werden in Kapitel 5 qualitative Erläuterungen zu den Erweiterungsmöglichkeiten und Grenzen von DigSim gegeben. Es folgen das Literaturverzeichnis und der vollständige Quellcode aller Dateien im Anhang. Ebenfalls im Anhang befindet sich die CD mit einer vollständigen Version von DigSim, mit einer interaktiven Dokumentation (inklusive Tex-File), mit Test-Files und mit allen Source-Files.

2 Benutzerhandbuch

Dieses Kapitel soll die Anwendung des Prgrammes DigSim erklären. Die Arbeitsumgebung, die Menus, die Eingabe werden diskutiert und in Kapitel 2.4 anhand eines Beispiels erläutert.

2.1 Programmstart

DigSim ist eine völlig unabhängige stand-alone Applikation, kann also einfach durch Doppelklick auf das Programmicon gestartet werden. Nachdem das Laufwerk mit dem Speicherplatz des Programms aufgerufen ist, öffnet man die Applikation wie gewohnt.

Es öffnet sich automatisch ein About-Dialog-Fenster mit einem einführenden grafischen Willkommensfenster. Dieses lässt sich durch Drücken des Schliessen-Buttons (ganz rechts oben im Fenster) oder auch einfach mit einem Mausklick auf irgendein Ort des Fensters schliessen. Man befindet sich nun in der Programmumgebung, das Transcript-Fenster ist geöffnet.
Man beachte, dass das Willkommens-Fenster nicht korrekt angezeigt wird, wenn die inkludierte Bilddatei sich nicht im selben Ordner wie die Applikation befindet.

2.2 Fenster

Zwei Arten von Fenstern werden von DigSim unterstützt. Sie regeln die Eingaben und strukturieren diese. Nach dem Programmstart ist automatisch das Transcript-Fenster aktiv und zur Eingabe bereit.

2.2.1 Transcript-Fenster

Das Transcript-Fenster entspricht eigentlich den Eingabefenstern von Programmen wie Scheme oder ähnlichen. Im Fenster können Ausdrücke direkt ausgewertet werden durch Drücken der Enter-Taste. Da ein metazirkulärer Evaluator steuert, der die gleiche Syntax wie Scheme behandelt, ist die normale Syntax von Scheme zu beachten. Das Transcript-Fenster dient vor allem auch dazu, die Resultate, die nach einer Ausführung eines digitalen Schaltkreises errechnet wurden, auszugeben. Das Transcript-Fenster ist also auch das Ausgabefenster aller Befehle und Eingaben in DigSim.

Abbildung 2 zeigt das Transcript-Fenster direkt nach dem Programmstart nach der Eingabe `(+ (* 3 4) 12)`. Es kann direkt nach dem Start von DigSim mit der Eingabe begonnen werden. Die Aufforderung *"Welcome, insert your commands please..."*, zu Beginn des Transcript-Fensters, erinnert daran.

2.2.2 Editier-Fenster

Durch den Menueintrag New oder den Shortcut Ctrl+N im Menu File gelangt man ins Editier-Fenster. Es öffnet sich automatisch nach der Befehlsausführung New. Ebenfalls öffnet sich automatisch ein solches Fenster, wenn

Abbildung 2: Das Transcript-Fenster nach der ersten Eingabe

eine bereits vorhandene Datei mittels dem Befehl Open geöffnet wird. Das Editier-Fenster hat zum Ziel, Schaltkreise in einer abstrakten Form zu modellieren und einzurichten, so dass diese auch als separate Projekte gespeichert und verwaltet werden können. Im Editier-Fenster wird eine Eingabe nicht nach Betätigen der Eingabetaste evaluiert. Es wird einfach eine neue Linie begonnen, auf die weitere Befehle eingegeben werden können. Es ist so möglich, einen ganzen Schaltkreis vorerst aufzubauen und erst am Ende der Modellierung den ganzen Schaltkreis auszuwerten.

Es können natürlich auch alle sonstigen Eingaben im Editier-Fenster vorbereitet werden (Abbildung 3), so dass auch diese erst durch eine spezielle Aufforderung ausgewertet werden. Erst nach einer expliziten Anweisung zur Evaluation können die abgebildeten Funktionen `proc1` und `proc2` im Transcript-Fenster aufgerufen werden. Diese quasi vordefinierten Funktionen im Editier-Fenster sind abspeicherbar in einer Datei, die den Inhalt des Editier-Fensters sichert. Dies erlaubt, bereits definierte Elemente immer wieder aufzurufen.

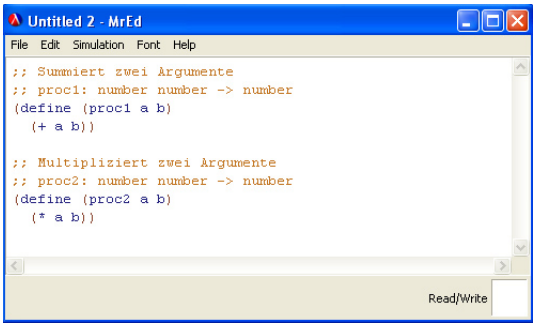


Abbildung 3: Im Editier-Fenster werden Eingaben nicht sofort ausgewertet

2.2.3 Menuleiste

Die einzelnen Menus in der Menuleiste von Transcript- und Editier-Fenster werden in diesem Unterkapitel kurz beschrieben.

Menu File: Mit *New* kann ein neues Editier-Fenster geöffnet werden. Dieses lässt sich nach getaner Arbeit mit *Safe* oder *Safe As* beliebig abspeichern. Ein bereits vorhandenes Projekt wird mit *Open* oder *Open Recent* geöffnet. Weiter können mit *Exit* und *Close* die Fenster wieder geschlossen werden. Verwendet man diese Befehle im Transcript, kann man das Programm verlassen.

Menu Edit: Dieses Menu entspricht den bekannten Edit Menus in diversen Programmen. Copy / Paste und andere Bearbeitungsfunktionen werden unter diesem Eintrag angeboten.

Menu Simulation: Mit *Eval Selection (Ctrl+M)* und *Eval Window (Ctrl+E)* können Teile des Editier-Fensters oder das gesamte Editier-Fesnter zur Auswertung übergeben werden. Die Resultate der Auswertung werden im Transcript gezeigt. Diese beiden Menueinträge sind sensitiv, das heisst, sie sind nur aktiv, wenn eine Markierung vorhanden ist (*Eval Selection*), oder wenn das Fenster nicht leer ist (*Eval Window*).
Reset Circuit (Ctrl+C) setzt die Agenda zurück und mit ihr alle Signale der definierten Drähte ebenfalls (alle Signale werden auf 0 gesetzt). Das heisst, dass nach *Reset Circuit* die aktuelle Zeit und die Drahtsignale wieder im Initiierungszustand sind. Mit dem Befehl *Reset (Ctrl+R)* wird das Transcript-Fenster wieder zurückgesetzt. Die globale Umgebung wird neu initiiert, was bedeutet, dass alle definierten Variablen gelöscht werden. Ein allfällig bereits evaluierter Schaltkreis steht so nicht mehr zur Verfügung und muss neu geladen werden.
Delays (Ctrl+D) öffnet eine grafische Eingabe, in der die Verzögerungen der primitiven Bauteile gesetzt werden können. Diese können natürlich auch mittels beispieliger Eingabe

```
(set! inverter-delay Wert)
(set! and-gate-delay Wert)
(set! or-gate-delay Wert)
```

gesetzt werden. *Propagation (Ctrl+P)* startet einen Durchlauf im digitalen Schaltkreis, startet quasi die Simulation. Mit dem letzten Eintrag *Get current time (Ctrl+T)* kann die gesamte bereits vergangene Zeitspanne ins Transcript ausgegeben werden.

Menu Font: Hier kann die Erscheinung der Eingabeschrift verändert werden. Es können so wichtige Sektionen von grossen Schaltkreisen beispielsweise speziell hervorgehoben werden.

Menu Help: Man gelangt zum About-Dialog, der auch beim Starten des Programms erscheint.

2.3 Beschreibung der Sprache

Die Sprache DigSim ist auf Scheme aufgebaut und deckt einen zusätzlichen Sektor, unseren spezifischen Sektor für digitale Schaltkreise, ab. Daraus folgt, dass viele Befehle von Scheme übernommen werden. Nur einige bestimmte Befehle wurden neu implementiert und erweitern die Sprache Scheme zu DigSim.

2.3.1 Befehle der Sprache Scheme

Folgende Befehle können analog zu Scheme verwendet werden. Diese werden im Source-File `MetaCircularEvaluator.scm` spezifiziert. Es werden folgende primitive Schemefunktionen unterstützt:

- *Listenoperatoren:*

  ```
  car, cdr, caar, cadr, cdar, cddr, caaar, caadr, cadar,
  cdaar, caddr, cddar, cdadr, cdddr, caaaar, caaadr,
  caadar, cadaar, cdaaar, caaddr, caddar, cddaar, cdaadr,
  cadadr, cdadar, cdddar, cadddr, cddddr, cons, null?,
  pair?, set-car!, set-cdr!, list, length, list-tail,
  list-ref, append, memq, memv, member, assq, assv, assoc
  ```

- *Boolsche Ausdrücke:*

  ```
  =, <, >, <=, >=
  ```

- *Rechenoperatoren und -funktionen:*

  ```
  +, -, *, /, remainder, modulo, exp, expt, random, eq?,
  eqv?, equal?, sin, cos, tan, asin, acos, atan, abs, not
  ```

- *Operationen mit Strings:*

  ```
  string->list, string->number, string->symbol,
  string-append, display
  ```

Weiter werden folgende Sonderformen auch in DigSim korrekt ausgewertet:

- `(define var expr)`, normale Definition einer Variablen
- `(set! var expr)`, normale Zuweisung
- `(define (var var1...) body)`, Funktionendefinition
- `(lambda (var1...) body)`, neutrale lambda-Funktion
- `(let ((var1 expr1) ...) body)`, syntaktischer Zucker für eine Anwendung einer lambda-Funktion
- `(let* ((var1 expr1) ...) body)`, Zuweisungen an lokale Variablen verlaufen chronologisch
- `(if condition then else)`, eine wenn - dann - sonst Klausel
- `(cond ([condition1 then1]... [else]))`, mehrere Klauseln
- `(begin expr1 expr2 ...)`, wertet mehrere Ausdrücke aus und liefert das Ergebnis des letzten Ausdrucks
- `'symbol`, Symbol Datentyp

Man erkennt, dass die gebräuchlichen Scheme Funktionen auch in DigSim verwendet werden können.

2.3.2 Befehle der Sprache DigSim

Für die Erweiterung zur Sprache DigSim wurden einige Komponenten und Variablen eingeführt. Diese dienen der Modellierung von Schaltkreisen und unterstützen diesen Prozess.
Folgende neue Funktionen und Variablen wurden eingeführt:

- inverter-delay: Zeitverzögerung eines inverter-gates, ist bei Programmstart auf 2 gesetzt.
  and-gate-delay: Zeitverzögerung eines and-gates, ist bei Programmstart auf 3 gesetzt.
  or-gate-delay: Zeitverzögerung eines or-gates, ist bei Programmstart auf 5 gesetzt.

- Funktionen für Drähte: Mit `(define a (make-wire))` wird a ein Draht zugewiesen. Syntaktischer Zucker für die gleichzeitige Defintion mehrere Drähte entspricht `(wires (wire1 wire2...))`. Damit auch lokale Drähte kreiert werden können, steht in Funktionen `(intern (wire1 wire2...))` zur Verfügung. Diese Prozedur weist `wire1 wire2...` Drähte zu, diese sind aber nur innerhalb der umgebenden Funktion definiert. Mit `(get-signal wire)` kann der Status eines Drahtes, also Null oder Eins, abgerufen werden. Um den Status eines Drahtes manuell zu verändern, wird `(set-signal! wire val)` verwendet. Es ist wichtig zu wissen, dass per default die Werte von Drähten mit Null definiert sind.

- Funktionen für die Modellierung von digitalen Schaltkreisen: Für die Definition von eigenen digitalen Bauteilen dient die Funktion `(defmudul name (input1 input2...) (output1 output2...) body)`. Mit `defmodul` wird ein Schaltelement `name` mit den gegebenen Inputs und Outputs definiert. Will man nun ein neu definiertes Schaltelement initialisieren, so kann mit `(inst name (input1 input2...) (output1 output2...))` ein solches erzeugt werden. Wie in der Einleitung erwähnt, bestehen alle digitalen Schaltkreise aus den drei primitiven Elementen and-gate, or-gate und Inverter. Diese drei Elemente sind in DigSim auch bereits vordefiniert, so dass sie für komplexere Schaltelemente mit Hilfe der Funktion `defmodul` erzeugt werden können.

- Simulation des Schaltkreises: Mit `(simulation)` wird eine standardisierte globale Umgebung mit den Standart-Zeitverzögerungen eingerichtet. Das heisst, der gesamte bis dahin ausgeführte Prozess wird auf den Anfangszustand zurückgesetzt. Damit ein Schaltkreis an bestimmten Schlüsselstellen auch überwacht werden kann, können Sonden `(probe 'symbol wire)` an Drähte angebracht werden. Diese Sonden melden sich automatisch, wenn der zu überwachende Draht seinen Zustand, das heisst seinen Wert, ändert. Es werden bei einer Änderung im Draht der Name der Sonde, spezifiziert durch `'symbol` in der Funktion `(probe symbol wire)`, der neue Wert und die aktuelle Zeit bei der

Werteänderung im Transcript ausgegeben. Die Simulation des Schaltkreises, ein neuer Durchgang, wird mit `(propagate)` aufgerufen. Die aktuelle Zeit kann laufend durch `(get-current-time)` aufgerufen und im Transcript-Window dargestellt werden.

2.4 Ein kleines Beispiel

Zur Illustration der Anwendungen mit DigSim wird im gezeigten Beispiel ein zusammengesetzter Schaltkreis, ein Halb-Addierer, implementiert. Es wird hierfür die Eingabe in einem neuen Editier-Fenster gebraucht. Abbildung 4 zeigt einen solchen Schaltkreis, der aus mehreren primitiven Bauteilen (siehe Abbildung 1) zusammengesetzt wird. Ein Halbaddierer verwertet zwei Dualzahlen und addiert diese. Ausgegeben werden zwei Output-Bits, die Summe und ein Carry-Bit, das einen allfälligen Übertrag anzeigt.

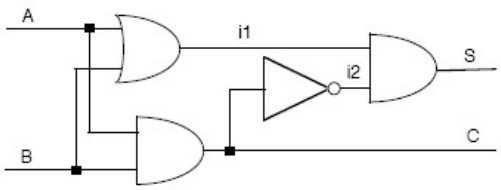


Abbildung 4: Halb-Addierer mit Drähten A und B als Inputs und S (Sum) und C (Carry-bit) als Outputs

Sonden sollen an den beiden Output-Drähten S und C angebracht sein, da diese interessant sind. Für die Propagation soll der Wert in Draht A auf Eins gesetzt werden. Dieses Projekt wird unter dem Namen halfadder gespeichert.

Abbildung 5 zeigt oben genanntes Projekt als Eingabe im Editier-Fenster. Das Fenster trägt bereits den Namen des Projekts, wurde also bereits zwischengespeichert. Nun soll unser Projekt ausgewertet werden. Hierfür wird *Eval Window* aus dem *Simulation* Menu gewählt. Die Eingaben wurden nun evaluiert. Damit die Sonden ihre Ausgaben tätigen, muss unser Schaltkreis propagiert werden. Man verwende hierzu den entsprechenden Menueintrag. Abbildung 6 zeigt die Ausgaben der Sonden im Transcript-Window.

Die ersten Ausgaben vor der Bestätigung *DONE - Eval Window* ergeben sich durch die Initialisierung des Halbaddierers. Da noch kein Durchlauf stattgefunden hat, ist auch noch keine Zeit vergangen und die Werte in den Drähten sind immer noch per default Null. Wird nun *(propagate)* aufgerufen, so wird unser Schaltkreis einmal durchlaufen. Die Ausgabe zeigt, dass der

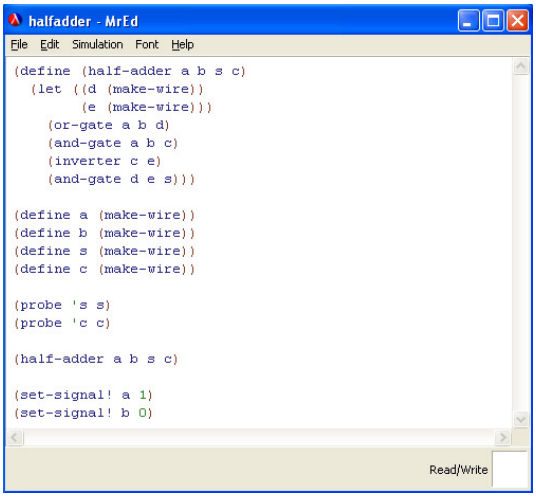


Abbildung 5: Editier-Fenster mit Halbaddierer als Projekt

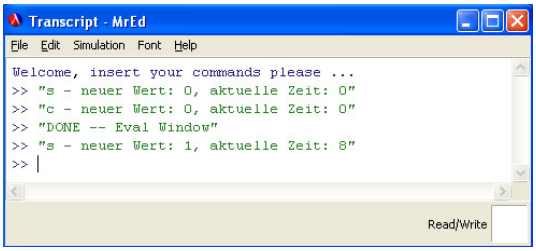


Abbildung 6: Ausgaben nach Eval Window und propagate im Transcript-Fenster

Draht mit der Sonde s, das heisst der Draht s, eine Zustandsänderung erfahren hat. Dies geschah nach 8 Zeiteinheiten (diese setzten sich aus den Default-Zeitverzögerungen der primitiven Bauteile zusammen). Der neue Wert des Drahtes ist nun Eins. Da die Binärzahlen 0 (Draht B) und 1 (Draht A) addiert die Summe 1 geben, ist die Änderung im Draht S, der für Summe steht, korrekt. Es gibt bei dieser Addition keinen Übertrag. Dies erklärt, warum die Sonde c im Draht C carry keine Zustandsänderung erfährt und deshalb nicht im Transcript-Fenster eine Ausgabe tätigt.

Das gleiche Beispiel des Halbaddierers kann auch mit Hilfe der in 2.3.2 erklärten Hilfsfunktionen etwas übersichtlicher und einfacher modelliert werden. Wir wollen mit Hilfe der Funktion `defmodul` zuerst einen solchen Halb-Addierer definieren und diesen danach in einer Instanz anwenden. Abbildung 7 zeigt das Editier-Fenster mit der Eingabe des halfadder2.

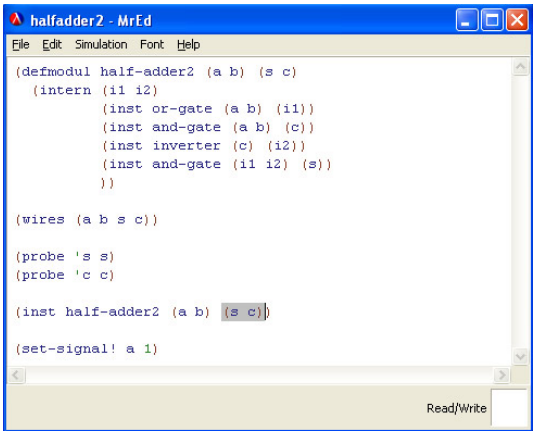


Abbildung 7: Projekt Halbaddierer mit Spezialfunktionen *defmodul*, *wires*, *intern* und *inst* definiert

2.5 Fehlermeldungen

Fehlermeldungen werden direkt ins Transcript-Fenster ausgegeben. Grundsätzlich kann bei Fehlermeldungen nicht auf den genauen Ort im Quelltext geschlossen werden. Die Meldung im Transcript-Fenster ist oft nicht aussagekräftig und man soll diese einfach als allgemeinen Fehler ansehen. Grundsätzlich fängt DigSim die gleichen Fehler ab wie Scheme. So wird eine falsche Syntax als Error-Meldung ausgegeben, ungebundene Variabeln werfen eine Ausgabe aus oder falsche Datentypen werden gemeldet. Wird keine Eingabe gemacht und trotzdem eine Evaluation gestartet, so wird eine nicht leere Eingabe gefordert. Wird die Auswertung einer primitiven Scheme-Prozedur verlangt, die in DigSim nicht implementiert ist, so wird ebenfalls eine Ausgabe ans Transcript-Fenster gesandt.

3 Programmiererhandbuch

In diesem Teil der Dokumentation über DigSim werden die einzelnen Komponenten des Quelltextes erklärt sowie ein Überblick über die Programmierideen gegeben. Es wird erläutert, wie der Source Code im Anhang gegliedert und organisiert ist, so dass der Leser weiss, wo er nach etwaigen Implementierungen und Komponenten suchen muss, falls er eine Änderung vornehmen will.

3.1 Organisation des Source Code

Der gesamte Quellcode wurde infolge besserer Lesbarkeit in mehrere einzelne Dateien unterteilt. Jede Datei enthält eine wichtige Komponente des Programms und ist in sich wie eine Implementierung einer Hauptkomponente von DigSim zu verstehen. Jede Datei beginnt mit einem Header, indem Autor, Datum, Referenzen und eine kurze Beschreibung des Zwecks des Files gegeben sind. Es folgt die Implementierung der Komponenten, welche mit Kommentaren vor einer Definition oder direkt im Quelltext einer Funktion unterstützt wird. Gegebenenfalls werden lange Files mit Untertiteln versehen und in Untersektionen gegliedert. Untersektionen beinhalten zu Beginn wiederum eine kurze Erläuterung der folgenden Prozeduren und Komponenten.

Die einzelnen Files mit ihrem Inhalt und Zweck werden im folgenden Abschnitt einzeln beschrieben:

myAboutDialog.scm: Das Startfenster beim Öffnen des Programms inklusive Steuerung des Schliessen per Mausklick sind in diesem File definiert. Man beachte, dass die gefertigte Bilddatei für den About-Dialog im selben Ordner wie diese Datei abgelegt sein muss. Für den programmierten Dialog muss die Datei `about_pic.jpg` vorhanden sein.

myMenus.scm: Trägt die neuen Menus *Simulation* und *Font* in die Menubar ein. Für das Simulation-Menu werden die vorhandenen Menueinträge mit Funktionen versehen, die den Befehl steuern. Das Menu Font wird standardmässig eingefügt. Ebenfalls sind in diesem File die on-demand Methoden für die Menueinträge Eval Window und Eval Selection definiert.

DelayDialog.scm: Baut eine grafische Oberfläche für das Setzen der Delays der primitiven Bauteile auf. Beim Aufruf des entsprechenden Menueintrages können neue Werte über diese Oberfläche gesetzt werden.

EnvironmentModel.scm: Das benötigte Umgebungsmodell für den Evaluationsprozess mit Bindungen, Rahmen und Umgebungen wird in diesem File implementiert. Diverse Funktionen steuern dabei eine korrekte Ablage der Daten im Sinne des Auswertungsprozederes im Umgebungsmodell.

Simulator.scm: Die Spezifikationen für die Modellierung von digitalen Schaltkreisen werden in diesem File beschrieben. Das Aussehen der primitiven Bauelemente oder von Verbindungsdrähten wird in `Simulator.scm` gezeichnet. Es sind die Erweiterungen für die Sprache DigSim also in diesem File zu suchen. Ebenfalls ist die Agenda

definiert, die den Zeitplan der zu tätigenden Aktionen verwaltet. Die Agenda beruft sich auf die Warteschlange, die mit `Queues.scm` gestützt wird.

Queues.scm: Hier sind Prozeduren definiert, die eine Warteschlange von Auswertungschritten beschreiben können. Eine Warteschlange definiert die Auswertungsreihenfolge im Schaltkreis und steuert so quasi den Durchlauf der Signale im Schaltkreis. Die Warteschlange wird von der Agenda benötigt, die die Inhalte der Warteschlange verarbeiten kann.

ReadEvalPrint.scm: Hauptsächlich sind in diesem File die Programmierungen zu finden, die das neue Interface von DigSim, das Transcript-Fenster mit Editier-Fenster definieren und steuern. Ein lesen-auswerten-ausgeben (read-eval-print) Kreislauf bearbeitet die Eingaben.

MetaCircularEvaluator.scm: Hier sind die Hauptprozeduren für die Auswertung von Ausdrücken in DigSim definiert. Die Hauptprozeduren `eval` und `apply` arbeiten koordiniert zusammen, um die Eingaben korrekt zu bearbeiten und die richtigen Resultate zu liefern.

standardMain.scm: Dies ist ein pseudo-File, das den Testzwecken während der Programmierung dient. Es werden die default-Verzögerungen gesetzt und die Ladevorgänge der einzelnen Dateien vorgegeben, so dass das Programm direkt in der Entwicklungsumgebung von Scheme getestet werden kann.

main.scm: Dies ist das Pendant zu `standardMain.scm`, das in der Sprache *module* geschrieben ist. Wiederum sind die Standard-Verzögerungen definiert und die Ladevorgänge der Files gegeben. Mit dieser Datei lässt sich nun eine völlig unabhängige stand-alone Applikation erzeugen. Es sind auch die nötigen Bibliotheken gegeben, die für ein funktionierendes Programm miteingefügt werden müssen. Man beachte, dass die Files in den Ladevorgängen alle im selben Ordner abgespeichert sein müssen, in dem auch das `main.scm`-File sich befindet. Weiter ist zu beachten, dass der Speicherpfad beim Erstellen der Applikation zum selben Ordner zeigt, in dem auch die einzubindenden Files abgelegt sind. Das Programm sollte sich am Schluss in einem Ordner befinden, in dem auch das Bild für den About-Dialog gespeichert ist. Ansonsten wird das Bild nicht gefunden und der Dialog nicht korrekt angezeigt.

3.2 Beschreibung der wichtigsten Komponenten von DigSim

3.2.1 Umgebungsmodell

Das Umgebungsmodell definiert die zu bearbeitende Datenstruktur des Evaluators. Es bestimmt beispielsweise die Darstellung von Prozeduren oder Umgebungen. Wie bereits beschrieben wurde, werden Zustände in Drähten durch Zuweisungen geändert `(set-signal! wire value)`. Man ist also zwingend auf ein Auswertungsmodell angewiesen, das mit Zuweisungen umgehen kann. Das in DigSim verwendete neue Auswertungsmodell besteht aus Umgebungen, in denen Zuweisungen an Variablen gemacht werden können. Eine Umgebung besteht aus einer Folge von Bindungsrahmen, die als eine Tabelle von Bindungen angesehen werden können. Eine Bindung bindet eine Variable an einen Wert. Von jedem Bindungsrahmen deutet ein Zeiger auf die zugehörige Umgebung. Weitere Erläuterungen zum Aufbau des Umgebungsmodells und zu den Auswertungsregeln in diesem werden in Kapitel 3.2 in [2] gemacht. Die Umgebungen, Rahmen und Bindungen wurden folgenderweise implementiert:

Umgebung: Eine Umgebung wird als Liste von Bindungsrahmen dargestellt. Im cdr der Liste ist die dazugehörende Umgebung abgespeichert. Als leere Umgebung wird einfach die leere Liste angenommen `(define empty-env '())`.

Rahmen: Liste von Bindungen `(list bindings)`.

Bindung: Pair von Variable und Wert `(cons var val)`.

Jeder Rahmen einer Umgebung ist also definiert als Liste von Bindungen, wobei eine Bindung als pair von Variable und Wert dargestellt wird. Wird eine bestimmte Variable gesucht, will man also den Wert einer bereits definierten Variabeln ausgeben, so wird im ersten Rahmen die passende Bindung gesucht. Wird diese im ersten Rahmen gefunden, so wird der Wert der Variablen ausgegeben. Sonst wird in der dazugehörenden, übergeordneten Umgebung nach der Variablen gesucht. Diese Schritte werden so oft wiederholt, bis man die entsprechende Bindung in einem Rahmen gefunden hat. Wird eine Zuweisung an eine Variable gemacht so wird gleich vorgegangen bis man die entsprechende Bindung gefunden hat. Danach wird durch die Zuweisung der Wert im pair der Bindung auf den neuen Wert gesetzt. Wird eine neue Variable definiert, so wird zuerst geschaut, ob bereits eine Variable gleichen Namens im ersten Rahmen besteht. Falls die Variable bereits existiert, wird die Bindung geändert. Sonst wird eine neue Bindung in den

Rahmen eingefügt. Für alle diese Suchoperationen in Umgebungen dient eine Hauptfunktion `(lookup-variable-value Variable Umgebung)` mit allen ihren Hilfsfunktionen.

3.2.2 Evaluator

Der in DigSim programmierte metazirkuläre Evaluator stützt sich auf das vorhin angeschnittene neue Umgebungsmodell der Auswertung. Durch die diesem Modell zugrunde liegenden Auswertungsregeln ergibt sich folgender Auswertungszyklus, zitiert aus Abelson 1996 [2], p. 378f:

> „... einen Grundzyklus, in dem Ausdrücke, die in Umgebungen ausgewertet werden sollen, auf Prozeduren reduziert werden, die auf Argumente angewendet werden sollen, die wiederum auf neue Ausdrücke reduziert werden, die in neuen Umgebungen ausgewertet werden sollen, und so weiter, bis wir bei Symbolen landen, deren Werte in der Umgebung nachgelesen werden, und elementaren Prozeduren, die direkt angewendet werden."

Der Evaluator wird diesem Grundzyklus mit den beiden Hauptfunktionen `eval` und `apply` gerecht. Durch die Interaktion dieser beiden Prozeduren wird genannter Auswertungszyklus angewandt.
Die Auswertungsprozedur `eval` ist als Fallunterscheidung zu sehen. Es werden die unterschiedlich syntaktisch möglichen Ausdrücke auf Art des Datentyps überprüft und an entsprechende Prozeduren weitergereicht. Die Anwendungsprozedur `apply` steuert die Anwendung von Prozeduren auf ein oder mehrere Argumente. Sie unterscheidet Prozeduren in zwei unterschiedliche Arten: Elementare Prozeduren und zusammengesetzte Prozeduren. Elementare Prozeduren werden direkt auf die Argumente angewandt, während zusammengesetzte Prozeduren angewendet werden, indem die Ausdrücke im Body der Prozedur nacheinander ausgewertet werden. Hierfür wird die Umgebungsstruktur entsprechend erweitert (Abelson 1996 [2], p. 380f).

Da der Evaluator letzten Endes Ausdrücke auf die Anwendung elementarer Prozeduren reduziert, müssen elementare Prozeduren von Scheme auch in DigSim Geltung haben und erkannt werden. Hierfür werden die Namen der elementaren Prozeduren als Bindungen in eine globale Umgebung gestellt, so dass `eval` bei der Auswertung einer Prozedur ein Objekt, das der elementaren Prozedur entspricht, in einer Umgebung findet und dieses an `apply` übergeben kann (Abelson 1996 [2], p. 398).

3.2.3 Simulator und Agenda

Im Simulator werden die wichtigsten Räpresentatoren von digitalen Schaltkreisen, die primitiven Bauelemente definiert. Sie regeln die Inputs von Drähten und können diese auf ihre spezifische Art verarbeiten und an den ausgehenden Output-Draht leiten. Weiter werden in den Schaltelementen die Verzögerungen bei einer Ausführung verarbeitet und verrechnet. Damit die Bauteile verbunden werden können, müssen Drähte modelliert werden, die ihren Status je nach Strom mit 1 oder 0 angeben. Dies geschah mittels einer Implementation von wires als Objekte. Jeder Draht ist somit eines dieser Objekte und erhält alle Grundfunktionen des Objektpakets. Zu einem Draht gehören die Sonden, die eine Ausgabe ans Transcript-Fenster senden, wenn der angeschlossene Draht seinen Wert geändert hat. Das System der Sonden ist ebenfalls im Simulator definiert.

Weiter wird im Simulator auch eine Grundsimulation definiert, mit derer sich die Grundumgebung installiert mit ihren Standarddaten für Delays etc.

Die Agenda entspricht einer Datenstruktur, welche einen genauen Zeitplan über die auszuführenden Aktionen enthält. DigSim als ein System, das Simulationen digitaler Logik durchführt, ist ein Beispiel für die so genannte ereignisgesteuerte Simulation. Das heisst, Ereignisse oder Aktivitäten stossen weitere Ereignisse an, die später in der Zeit liegen und ebenfalls neue Aktivitäten initiieren (vergleiche den Durchgang eines Signals durch einen digitalen Schaltkreis). Die Agenda kann als Buch angesehen werden, in dem diese Kettenreaktionsreihen jeweils eingetragen werden und Schritt für Schritt abgearbeitet werden. Anders gesagt, sind in der Agenda die auf ihre Auswertung wartenden Prozeduren chronologisch eingetragen (Abelson 1996 [2], p. 284f).

Eine Agenda besteht aus Zeitsegmenten, wobei ein Zeitsegment ein pair mit der Zeit und einer Warteschlange als Elemente ist. Eine Warteschlange *(engl. queue)* beinhaltet die Prozeduren (die Aktivitäten), die während dieses Zeitsegments ausgeführt werden müssen.

Die Datenstruktur der Agenda wurde nach folgendem Muster implementiert:
Eine Agenda ist eine Liste, in der das erste Element der aktuellen Zeit entspricht und deren cdr Zeitsegmenten entspricht. Die leere Agenda entspricht einer Liste mit der aktuellen Zeit am Kopf und beinhaltet noch keine Zeitsegmente. Die Agenda steuert die Zeitsegmente und die Chronologie wie auf Seite 295 in [2] zitiert:

> „Um einen Vorgang in die Agenda einzutragen, prüfen wir zuerst, ob die Agenda leer ist. Wenn das der Fall ist, erzeugen wir

> ein Zeitsegment für den Vorgang und installieren dieses in der Agenda. Andernfalls durchsuchen wir die Agenda, wobei wir die Zeit eines jeden Segments prüfen. Wenn wir ein Segment mit unserer festgelegten Zeit finden, tragen wir unseren Vorgang in die dazugehörige Warteschlange ein. Wenn wir auf einen Zeitpunkt treffen, der nach unserer festgelegten Zeit liegt, müssen wir direkt davor ein neues Zeitsegment in die Agenda eintragen. Wenn wir das Ende der Agenda erreichen, müssen wir dort ein neues Zeitsegment eintragen."

Die Datenstruktur und die Prozeduren, die die zitierte Steuerung modellieren, sind im Quellcode der beiden Files `Queues.scm` und `Simulator.scm` ersichtlich.

4 Ausblick und Beurteilung des Projektes

4.1 Grenzen von DigSim

DigSim ist in seiner Anwendung und in seinen Möglichkeiten Fehler abzufangen beschränkt.

Es kann der Fall auftreten, dass aus Unachtsamkeit ein Input eines primitiven Gates dem Output desselben Gates entspricht. Das heisst, man hat mit demselben Draht ein primitives Bauteil gespeist, welcher auch als Ausgabe an das Bauteil programmiert wurde. Beispielsweise wäre

```
(inst and-gate (Draht1 Draht2) (Draht1))
```

ein solcher Fall. Natürlich würde hier eine endlose Schlaufe produziert, die nur durch einen Systemunterbruch abzubrechen wäre. Solche Schlaufen werden von DigSim nicht gemeldet und bei einer allfälligen Propagation würde sich das Programm in einer Endlosschlaufe aufhängen.

Wird ein Fehler im Transcript-Fenster gemeldet, so ist dieser meist nicht vollständig selbsterklärend. Man kann oft aus den Fehlermeldungen den genauen Ort des Fehlers nicht ableiten. Auch die ausgebende Fehlerprozedur wird nicht unbedingt mit der Fehlermeldung mitgeliefert.

Für sehr grosse Projekte und komplizierte Schaltkreise eignet sich DigSim nicht unbedingt. Will man grosse Schaltkreise korrekt untersuchen und beobachten, so braucht man mehrere Sonden, die ihrerseits ständig bei einem Wechsel des Signals eine Ausgabe produzieren. Bei vielen Sonden wird so die Resultatausgabe ziemlich unübersichtlich. Das Nachvollziehen eines Durchlaufes (Propagation) durch den Schaltkreis würde durch den Kommentar

unzähliger Sondenausgaben schwieriger.
Ein weiterer Nachteil kommt ebenfalls bei grossen Projekten zum Tragen: Grosse Projekte überfordern relativ schnell das System. Die Rechenzeit eines Durchlaufes nimmt nicht linear zu, weshalb schnell eine Schwelle erreicht ist, bei der das Programm an seine Grenzen stösst.

Die Modellierung eines Schaltkreises erfolgt in DigSim auf eine ungewöhnliche, abstrakte Weise. Man ist vor einem ersten Gebrauch auf ein Tutorial wie diese Dokumentation angewiesen, damit man weiss, auf welche Art und Weise ein Schaltkreis im Programm aufgebaut wird. Hält man sich nicht an diese abstrakten Modellierungsgrundlagen, so kommt man nicht zum Ziel.
Auch zur Deutung der Sondenausgaben bedarf man einer ersten Erklärung. Nur so ist eine genaue Übersicht über die diversen chronologischen Ausgaben der Sonden möglich.

Allgemein ist das Programm eher abstrakt gehalten. Der Benutzer muss alle seine Befehle und Wünsche auch selbst initiieren, auch wenn ein bereits beabsichtigter Vorgang erkennbar ist. Beispielsweise muss nach einer Änderung der Delays manuell ein neuer Durchlauf initiiert werden, obwohl dies nach einer Änderung wohl beabsichtigt sein wird. Diese statische Grundform des Programms erlaubt es aber dem Benutzer, seine Befehle sehr genau zu steuern und zu überblicken. Keine automatischen Vorgänge, die undurchsichtig sind, lotsen in die Ungewissheit. Die Eingaben sind immer klar Schritt für Schritt zu tätigen.

Würde eine Ausgabe für Test- oder Entwicklungszwecke eines Schaltkreises gebraucht, so müsste eine Schnittstelle der Ausgabe zu einem Drucker installiert werden. Die Ausgaben müssten tabellarisch angeordnet sein und in ein entsprechend mobiles Dateiformat konvertierbar sein müssen. Nur so wäre auch ein tatsächlicher Nutzen eines solchen Programms zu erreichen - abgesehen natürlich von zusätzlicher Performance und Bedienerfreundlichkeit.

4.2 Realisierte Erweiterungen

Das hier beschriebene Projekt enthält alle Grundvoraussetzungen und nur kleinere Erweiterungen zu den gegebenen Punkten für das Projekt. Das Augenmerk wurde auf die Grundfunktion des Programms geworfen, so dass für grosse Erweiterungen keine Zeit übrig geblieben ist.

- Realisiert wurde eine kleine Hilfe, die aktuelle Zeit auszugeben. Durch einen Menupunkt im Menu Simulation kann man die aktuelle Zeit der

Agenda ins Transcript-Fenster drucken, so dass man den Überblick über die Durchlaufszeit immer direkt ausgeben kann.

- Dem Benutzer stehen zwei Reset-Möglichkeiten zur Verfügung. Einerseits kann er durch *Reset* quasi das Programm in den Startzustand zurücksetzen. Will er jedoch nur den digitalen Schaltkreis, den er bereits aufgebaut und evaluiert hat, in den Anfangszustand versetzen, so steht ihm mit dem Menueintrag *Reset Circuit* ein entsprechendes Tool zur Verfügung.

- Weiter wurden die im Menu "Simulation" beinhalteten Menuitems "Eval Selection" und "Eval Window" sensitiviert. Das heisst, dass ihnen mit einer on-demand Methode eine Funktion mitgegeben wird, die testet, ob dieser Menueintrag überhaupt aktiv sein sollte. Ist beispielsweise kein Text markiert, ist der Befehl "Eval Selection" überflüssig. Ebenfalls macht "Eval Window" keinen Sinn, wenn das Fenster leer ist. Die on-demand Funktion wird vor jedem Menuaufruf in der Menuleiste gestartet.
  Zur weiteren Einschränkung wird durch die mitgegebene Funktion auch getestet, ob man sich im Transcript-Fenster befindet. Ist dies der Fall, so machen weder "Eval Selection" noch "Eval Window" Sinn und die beiden Menuitems sind passiv.

4.3 Weitere Erweiterungsmöglichkeiten

DigSim liesse sich beinahe uneingeschränkt erweitern. Die Anwendung könnte durch unzählige Erweiterungen zu einem bedienerfreundlichen (Markt-) Produkt ausgebaut werden.

Um das Spektrum an Möglichkeiten der Erweiterung aufzuzeigen sind nachfolgend einige Ideen aufgezeigt:

- Um eine bessere Lesbarkeit von Fehlermeldungen zu kriegen, könnten diese so angepasst werden, dass das error-handling mit seinen Ausgaben selbsterklärend würde. Dies würde den Benutzer in seinen Eingaben besser lenken und allfällige Probleme mit DigSim eher lösen.

- Das Bild, das beim Programmstart auf dem Bildschirm erscheint, das Willkommensfenster, sollte unabhängig sein von der Bilddatei des About-Dialoges. Es ist hinderlich, wenn das Programm nicht nur aus einer einzigen ausführbaren Datei besteht und immer noch die Bilddatei mit in denselben Ordner kopiert werden muss. Die Bilddatei müsste

im `main.scm` file mitgegeben werden können, so dass die Applikation unabhängig von der Bilddatei würde.

- Die Ausgabe der Sonden bei einem Durchlauf scheint etwas unkoordiniert auf dem Transcript-Fenster ausgegeben zu werden. Eine Erweiterung, die die Ausgabedrähte mit ihren Signaländerungen in einer grafischen Tabelle zeigen würde, würde eine bessere Lesbarkeit und Übersicht bringen. Auch könnte man so die Signaländerungen in den Sonden so filtern, dass nur noch die letzte Änderung in der Sonde ausgegeben würde.

- Für einen schnelleren Aufbau von Schaltkreisen könnte man die Liste mit primitiven Elementen auf die häufigsten verwendeten Elemente erweitern. Wären diese bereits in der Standardapplikation vorhanden, liessen sich kompliziertere Schaltkreise schneller aufbauen und modellieren. Diese neuen Bauelemente müssten im Handbuch genau erklärt werden, damit sie auch genutzt werden könnten.

- Was die Anwendung fast marktreif machen würde, wäre eine grafische Unterstützung für den Modellierungsprozess mit den vordefinierten Bauelementen. Man könnte die Bauelemente auf dem Bildschirm auf eine Position setzen, diese mit Drähten mit andern Elementen verbinden und Sonden an Drähte setzen. Es könnte der gesamte Schaltkreis analog zu einer Zeichnung auf dem Bildschirm mit der Maus aufgebaut werden und im Hintergrund in DigSim übersetzt werden. Es würde so das grösste Hindernis für den Benutzer, die abstrakte Modellierungsart würde entfallen und liesse auch einem Neunutzer von DigSim eine rasche und richtige Benutzung zu.

- Ein genaueres und ausgebautes Hilfemenu würde den Benutzer besser unterstützen und leiten. Eine Indexsuche in einem Hilfetext, analog zum Kapitel Benutzerhandbuch, würde die Hilfesuche verschnellern und vereinfachen.

Literatur

[1] Harold Abelson/ Gerald Jay Sussman/ Julie Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, second edition, 1996.

[2] Harold Abelson/ Gerald Jay Sussman/ Julie Sussman. *Struktur und Interpretation von Computerprogrammen*. Springer-Verlag Berlin Heidelberg, Zweite Edition, 1998.

A Source Code

A.1 Datei “myAboutDialog.scm”

```
; ------------------------------------------------------------------------
; PROJEKT INFORMATIK IA, 2004/2005
; UNIVERSITAET FREIBURG_____________________________________________________


; Autor:    Urs Sieber
; Datum:    27.12.2004
; Filename: myAboutDialog.scm


; ************************************************************************
; DESCRIPTION: Set screen variables


(define screen-width 0)
(define screen-height 0)

(define (update-screen-variables)
  (let-values (((width height) (get-display-size)))
    (set! screen-width width)
    (set! screen-height height)))

; ************************************************************************
; DESCRIPTION: Execute update-screen-variables

(update-screen-variables)


; ************************************************************************
; DESCRIPTION: About Dialog Window

(define aboutDialogPicture "about_pic.jpg")

(define pathToDialogPic
  (let* ((pRun (build-path (current-directory) aboutDialogPicture))
         (pTmp (path-only (find-system-path 'exec-file)))
         (pExec (build-path  pTmp aboutDialogPicture)))
    (if (file-exists? pRun) pRun
        (if (eq? (system-type) 'macosx)
            (build-path pTmp  'up 'up 'up aboutDialogPicture)
            pExec))))

(define AboutBitmap
    (instantiate bitmap% (pathToDialogPic)))

(define (paint-proc theCanvas theDc)
  (send theDc draw-bitmap AboutBitmap 0 0))

(define AboutDialog
  (instantiate dialog%
    ("Welcome to Project digSIM" #f #f #f
      (floor (- (/ screen-width 2) (/ (send AboutBitmap get-width) 2)))
      (floor (- (/ screen-height 2) (/ (send AboutBitmap get-height) 2)))
      )))

(define ImageCanvas%
  (class canvas% ()
```

```
    (override on-event)
    (define (on-event ev)
      (if (send ev get-left-down)
          (send (send this get-parent) on-exit)))
    (super-instantiate ())))

(define theCanvas
  (instantiate ImageCanvas% (AboutDialog '() paint-proc)
    (min-width (send AboutBitmap get-width))
    (min-height (send AboutBitmap get-height))))

; ************************************************************************
; DESCRIPTION: Function to call the About-Dialog (send AboutDialog show #t)
; An on-load-delay should rule the loading process in the manner that the
; about-picture will never be unloaded.

(define (show-about-dialog)
  (sleep/yield 0.1)
  (send AboutDialog show #t))
```

A.2 Datei "myMenus.scm"

```
; ------------------------------------------------------------------------
; PROJEKT INFORMATIK, 2004/05
; UNIVERSITAET FREIBURG_____________________________________________


; Autor:    Urs Sieber
; Datum:    29.12.2004
; Filename: myMenus.scm


; ************************************************************************
; DESCRIPTION: Callbacks für Menueinträge

;(define (callback-sim Item)
;  (print-to-transcript Item))

(define (menuItemEvalSelection MItem CEvent)
  (with-handlers ([exn? (lambda (exn) (print-error-msg (exn-message exn)))])
  (let ((theWindow (get-top-level-focus-window)))
    (let* ((ed (send theWindow get-editor))
           (start (send ed get-start-position))
           (end (send ed get-end-position))
           (contents (send ed get-text start end #f #f)))
      (unless (= (string-length contents) 0)
        (let ((theList (extended-read contents)))
          (eval-sequence theList the-global-environment)
          (print-to-transcript "DONE -- Eval Selection")
          (transcript_prompt)))))))

(define (on-demand-selection MItem)
       (let ((theWindow (get-top-level-focus-window)))
         (let* ((ed (send theWindow get-editor))
                (name (send theWindow get-label))
                (start (send ed get-start-position))
                (end (send ed get-end-position)))
           (if (equal? name "Transcript")
               (send MItem enable #f)
               ;else-if
                 (if (= start end)
```

```
                    (send MItem enable #f)
                    (send MItem enable #t)))
          )))

(define (menuItemEvalWindow MItem CEvent)
  (with-handlers ([exn? (lambda (exn) (print-error-msg (exn-message exn)))])
    (let ((theWindow (get-top-level-focus-window)))
      (unless (equal? "Transcript" (send theWindow get-label))
        (let* ((ed (send theWindow get-editor))
               (contents (send ed get-text 0 'eof #f #f))
               (contents-length (string-length contents)))
          (unless (= contents-length 0)
                (let ((theList (extended-read contents)))
                  (eval-sequence theList the-global-environment)
                  (print-to-transcript "DONE -- Eval Window")
                  (transcript_prompt))))))))

(define (on-demand-window MItem)
       (let ((theWindow (get-top-level-focus-window)))
         (let* ((ed (send theWindow get-editor))
                (name (send theWindow get-label))
                (start (send ed get-start-position))
                (end (send ed get-end-position))
                (contents (send ed get-text 0 'eof #f #f))
                (contents-length (string-length contents)))
           (if (equal? name "Transcript")
               (send MItem enable #f)
               ;else-if
                 (if (= contents-length 0)
                     (send MItem enable #f)
                     (send MItem enable #t)))
           )))

(define (menuItemReset MItem CEvent)
  (simulation)
  (set! the-global-environment (setup-environment))
  (send TranscriptEditor erase)                 ; Delete transcript content
  (print-to-transcript "DONE -- Reset")         ; Confirmation
  (transcript_prompt)                           ; newline
  )

(define (menuItemResetWires MItem CEvent)
  (set! the-agenda (make-agenda))
  (reset-wire-queue my-wire-queue)
  ;(send TranscriptEditor erase)                ; Delete transcript content
  (print-to-transcript "DONE -- Reset Circuit") ; Confirmation
  (transcript_prompt)                           ; newline
  )

(define (menuItemDelays MItem CEvent)
  (callback-delay))

(define (menuItemPropagation MItem CEvent)
  (propagate))

(define (menuItemTime MItem CEvent)
  (get-current-time))


; ***************************************************************************
; DESCRIPTION: Menuitems for Simulation-Menu
```

```
(define (make-simulation-menu-items Menu)
    (instantiate menu-item%
      ("Eval Selection" Menu menuItemEvalSelection #\m #f on-demand-selection))
    (instantiate menu-item%
      ("Eval Window" Menu menuItemEvalWindow #\e #f on-demand-window))
    (instantiate menu-item% ("Reset" Menu menuItemReset #\r))
    (instantiate menu-item% ("Reset Circuit" Menu menuItemResetWires #\c))
    (instantiate separator-menu-item% (Menu))
    (instantiate menu-item% ("Delays" Menu menuItemDelays #\d))
    (instantiate menu-item% ("Propagation" Menu menuItemPropagation #\p))
    (instantiate menu-item% ("Get current time" Menu menuItemTime #\t)))


; ***************************************************************************
; DESCRIPTION: Added Menus for Menu-bar

(define (add-menus frame)
  (let* ((myMenubar (send frame get-menu-bar))
         (simulationMenu (instantiate menu% ("Simulation" myMenubar)))
         (setup-font-menu (instantiate menu% ("Font" myMenubar))))
    (make-simulation-menu-items simulationMenu)
    (append-editor-font-menu-items setup-font-menu)
    (frame:reorder-menus frame)
    ))
```

A.3 Datei “DelayDialog.scm”

```
; ---------------------------------------------------------------------------
; PROJEKT INFORMATIK, 2004/05
; UNIVERSITAET FREIBURG_____________________________________________________

; Autor:    Urs Sieber
; Datum:    22.03.2005
; Filename: DelayDialog.scm

; ***************************************************************************
; DESCRIPTION: set-delays sets the global variables of delay.

(define (set-delays list)
  (set-variable-value! 'inverter-delay
    (string->number (caddr list)) the-global-environment)
  (set-variable-value! 'and-gate-delay
    (string->number (car list)) the-global-environment)
  (set-variable-value! 'or-gate-delay
    (string->number (cadr list)) the-global-environment)
)



; ***************************************************************************
; DESCRIPTION: Opens a dialog-window where the set the delay variables as
; input.

(define (callback-delay)
 (let* ((theDialog (make-object dialog% "Delays" #f 200 #f 100 100))
        (theButton 'none)
        (f1 (make-object text-field% "AND-Gate Delay:   " theDialog void
         (number->string
          (lookup-variable-value 'and-gate-delay the-global-environment))))
        (f2 (make-object text-field% "OR-Gate Delay:     " theDialog void
         (number->string
```

```
          (lookup-variable-value 'or-gate-delay the-global-environment))))
        (f3 (make-object text-field% "INVERTER Delay:   " theDialog void
         (number->string
          (lookup-variable-value 'inverter-delay the-global-environment))))

        (p1 (make-object horizontal-panel% theDialog))
        (b1 (make-object button% "Cancel" p1 (lambda (b c)
                                               (set! theButton 'cancel)
                                               (send theDialog show #f))))
        (b2 (make-object button% "OK" p1 (lambda (b c)
                                           (set! theButton 'ok)
                                           (send theDialog show #f))
             '(border))))
   (send f1 vert-margin 10)
   (send f2 vert-margin 10)
   (send f3 vert-margin 10)
   (send p1 vert-margin 20)
   (send p1 set-alignment 'center 'center)
   (send theDialog show #t)
   (if (eq? theButton 'ok)
       (set-delays
        (list (send f1 get-value) (send f2 get-value) (send f3 get-value)))
       )
  )
 )
```

A.4 Datei "EnvironmentModel.scm"

```
; ---------------------------------------------------------------------
; PROGRAMMIERUNGI: PROJEKT, 2004/2005
; UNIVERSITAET FREIBURG_____________________________________________


; Autoren:  Urs Sieber
; Datum:    22.03.2005
; Filename: EnvironmentModel.scm

; ********************************************************************




; *********** SHORT DESCRIPTION OF THE PURPOSE OF THIS FILE *************
;
; Definition of some routines for implementing the runtime-environment
; of the evaluator.
; A description of another implementation of the functions below is given
; in the book 'Structure and Interpretetion of Computer Programs'
; of Abelson & Sussman in the Section 4.1.3.
;
; ********************************************************************


#| ---------------------------------------------------------------------

   The environment model, as implemented in this file:

    - An environment is a list of frames. The enclosing environment of an
      environment is the cdr of the list. The empty environment is simply
      the empty list.
```

```
    - Each frame of an environment is represented as list of bindings

    - A binding is a pair of <var> and <value>

    - To look up a variable in an environment, we scan the list of variables
      in the first frame. If we find the desired variable, we return the
      corresponding value. If we do not find the variable in the current
      frame, we search the enclosing environment, and so on. If we reach the
      empty environment, we signal an "unbound variable" error.

    - To set a variable to a new value in a specified environment, we scan
      for the variable, just as in look-up-variable-value, and change the
      corresponding value when we find it.

    - To define a variable, we search the first frame for a binding for the
      variable, and change the binding if it exists. If no such binding
      exists, we adjoin one to the first frame.

   ------------------------------------------------------------------------
|#




; DESCRIPTION: Displays an error message
; CONTRACT: error: Strings or numbers or symbols -> string
; EXAMPLE: (error 'urs 'tobias 123 123 "String1" "String2")
; RESULT:  (urstobias123123String1String2)

#|
(define (error . msg)
  (map display msg)
  (newline))
|#


;***************************************************************************
; Data Definition Binding:
; A Binding is a pair of variable and value
;***************************************************************************
; DESCRIPTION: Constructor of a binding
; CONTRACT: make-binding: variable X -> binding
(define (make-binding variable value)
  (cons variable value))

; DESCRIPTION: Gives the variable name of a binding
; CONTRACT: binding-variable: binding -> variable
(define (binding-variable binding)
  (car binding))

; DESCRIPTION: Gives the value of a binding
; CONTRACT: binding-value: binding -> X
(define (binding-value binding)
  (cdr binding))

; DESCRIPTION: Sets a new value for a preexisting binding
; CONTRACT: set-binding-value!: binding X -> void
(define (set-binding-value! binding value)
  (set-cdr! binding value))
```

```
;*****************************************************************************
; Data Definition Frame:
; A Frame is a list of bindings
;*****************************************************************************

; DESCRIPTION: Constructor of a frame
; CONTRACT: make-frame: list-of-variables list-of-values -> frame
(define (make-frame variables values)
  (cond
    [(and(null? variables)(null? values))'()]
    [(null? variables)(error 'make-frame "\nDie Variable ist NULL")]
    [(null? values)(error 'make-frame "\nDas Value ist NULL")]
    [else
     (cons (make-binding(car variables)(car values))
           (make-frame (cdr variables)(cdr values)))]))

; DESCRIPTION: Adds a binding to an existing frame
; CONTRACT: adjoin-binding: binding frame -> frame
(define (adjoin-binding binding frame)
  (cons binding frame))

; DESCRIPTION binding-in-frame: Looks up, if a binding is still existing
; in a frame
; CONTRACT: binding-in-frame: variable frame -> binding or no-binding
; DESCRIPTION assq: Helpfunction for binding-in-frame
; CONTRACT: assq: variable frame -> binding or no-binding
; The function assq is local defined, because it still exists as
; Scheme primitive.

(define (binding-in-frame var frame)
  (define (assq key bindings)
  (cond
    [(null? bindings) no-binding]
    [(eq? key (binding-variable (car bindings))) (car bindings)]
    [else
     (assq key (cdr bindings))]))
  (assq var frame))

; DESCRIPTION: Looks up if a binding was found
; CONTRACT: found-binding?: boolean -> boolean
(define (found-binding? b)
  (not(eq? b no-binding)))

; DESCRIPTION: Definition of no-binding
(define no-binding #f)




;*****************************************************************************
; Data Definition Environment:
; An Environment env is a list of frames
;*****************************************************************************

; DESCRIPTION: Gives the first Frame of an Environment
; CONTRACT: first-frame: env -> frame
(define (first-frame env)
  (car env))

; DESCRIPTION: Gives the rest of Frames of an Environment
```

```
; CONTRACT: rest-frame: env -> env
(define (rest-frames env)
  (cdr env))

; DESCRIPTION: Looks up if the env is empty
; CONTRACT: no-more-frames?: env -> boolean
(define (no-more-frames? env)
 (null? env))

; DESCRIPTION: Adds a Frame to an Environment
; CONTRACT: adjoin-frame: frame env -> env
(define (adjoin-frame frame env)
  (cons frame env))

; DESCRIPTION: Sets the first Frame of an Environment
; CONTRACT: set-first-frame!: env frame -> void
(define (set-first-frame! env new-frame)
  (set-car! env new-frame))


;**************************************************************************
; Operations on environments
;**************************************************************************

; DESCRIPTION: Adds a frame to an environment, with the values and the
; variables of the new frame.
; CONTRACT: extend-environment: list-of-variables list-of-values env -> env
(define (extend-environment variables values base-env)
  (if (= (length variables)(length values))
      (adjoin-frame (make-frame variables values) base-env)
      (if (< (length variables)(length values))
          (error 'extended-environment
                 "\nToo many arguments supplied" variables values)
          (error 'extended-environment
                 "\nTo few arguments supplied" variables values))))

; DESCRIPTION: Looks for a variable in an environment and sets their value.
; CONTRACT: set-variable-value!: variable X env -> void
(define (set-variable-value! var val env)
  (let((b (binding-in-env var env)))
    (if(found-binding? b)
       (set-binding-value! b val)
       (error 'set-variable-value! "\nError setting the Variable " var "
                                    - Your Variable is unbounded"))))

; DESCRIPTION: Defines a variable in an environment.
; If the variable already exists, the value is reseted, else a new binding
; will be installed.
; CONTRACT: define-variable!: variable X env -> env
(define (define-variable! var val env)
  (let ((b (binding-in-frame var (first-frame env))))
    (if (found-binding? b)
        (set-binding-value! b val)
        (set-first-frame!
         env
         (adjoin-binding(make-binding var val)
                        (first-frame env))))))

; DESCRIPTION: Searches a binding in an Environment and if the variable
; exists it returns the binding else a #f.
; CONTRACT: binding-in-env: variable env -> binding or no-bindung
(define (binding-in-env var env)
```

```
  (if (no-more-frames? env)
      no-binding
      (let ((b (binding-in-frame var (first-frame env))))
      (if (found-binding? b)
          b
          (binding-in-env var (rest-frames env))))))

; DESCRIPTION: Looks for a variable in every frame of an environment.
; If it is founded, the binding will be the output,
; else an error-message occurs.
; CONTRACT: define-variable!: variable env -> binding or no-bindung
(define (lookup-variable-value var env)
  (let((b (binding-in-env var env)))
    (if (found-binding? b)
        (binding-value b)
        (error 'lookup-variable-value
         "\nYour argument is unbounded! Variable " var " was not found."))))

; DEFINITION OF THE EMPTY ENVIRONMENT:
(define the-empty-environment '())




;**************************************************************************
; ABSTRACT: Here are some exemples which can be used to test the
;           implementation of the environment model.
;**************************************************************************
;
;           The following examples are used below:
;
;               1. a)  (extend-environment '(a x) '(1 2) '())
;               2. a)  (lookup-variable-value 'a the-global-environment)
;                  b)  (lookup-variable-value 'x the-global-environment)
;                  c)  (lookup-variable-value 'z the-global-environment)
;               3. a)  (define-variable! 'b 3 the-global-environment)
;                  b)  (define-variable! 'a 4 the-global-environment)
;               4. a)  (extend-environment '(a y) '(5 6)
;                         the-global-environment)
;                  b)  (lookup-variable-value 'a NewEnv)
;                  c)  (lookup-variable-value 'x NewEnv)
;               5. a)  (set-variable-value! 'a 7 NewEnv)
;                  b)  (set-variable-value! 'x 8 NewEnv)


#|
;; #########################################################################
;; EXAMPLE No. 1 : The function 'extend-environment'...
(define the-global-environment (extend-environment '(a x) '(1 2) '()))
the-global-environment
;(((a . 1) (x . 2)))



;; #########################################################################
;; EXAMPLE No. 2 : The function 'lookup-variable-value'...
(lookup-variable-value 'a the-global-environment)
;1
(lookup-variable-value 'x the-global-environment)
;2
(lookup-variable-value 'z the-global-environment)
;Unbound variable z
```

```
;; ######################################################################
;; EXAMPLE No. 3 : The function 'define-variable!'...
(define-variable! 'b 3 the-global-environment)
the-global-environment
;(((b . 3) (a . 1) (x . 2)))
(define-variable! 'a 4 the-global-environment)
the-global-environment
;(((b . 3) (a . 4) (x . 2)))



;; ######################################################################
;; EXAMPLE No. 4 : Environment with two frames...
(define NewEnv (extend-environment '(a y) '(5 6) the-global-environment))
the-global-environment
;(((b . 3) (a . 4) (x . 2)))
NewEnv
;(((a . 5) (y . 6)) ((b . 3) (a . 4) (x . 2)))
(lookup-variable-value 'a NewEnv)
;5
(lookup-variable-value 'x NewEnv)
;2



;; ######################################################################
;; EXAMPLE No. 5 : The function 'set-variable-value!'...
(set-variable-value! 'a 7 NewEnv)
NewEnv
;(((a . 7) (y . 6)) ((b . 3) (a . 4) (x . 2)))
the-global-environment
;(((b . 3) (a . 4) (x . 2)))
(set-variable-value! 'x 8 NewEnv)
NewEnv
;(((a . 7) (y . 6)) ((b . 3) (a . 4) (x . 8)))
the-global-environment
;(((b . 3) (a . 4) (x . 8)))
|#
```

A.5 Datei "Simulator.scm"

```
; ------------------------------------------------------------------------
; PROJEKT INFORMATIK, 2004/05
; UNIVERSITAET FREIBURG------------------------------------------------------


; Autor:    Urs Sieber
; Datum:    24.03.2005
; Filename: Simulator.scm



; ************** SHORT DESCRIPTION OF THE PURPOSE OF THIS FILE ****************

; The code is taken from the following book:

;   - Book    : Structure and Interpretetion of Computer Programs
;   - Edition : Second Edition
```

```
;   - Autor   : Harold Abelson & Gerald Jay Sussman
;   - Section : 3.3.4 A Simulator for Digital Circuits
;   - Pages   : 273 to 285

; ****************************************************************************


; The contents of the file "Queues.scm" is needed for the agenda...
;(load "Queues.scm")



; ############################################################################
; SECTION 3.3.4 A Simulator for Digital Circuits
; ############################################################################


; ============================================================================
; Primitive Function Boxes - primitive gates offered by the simulator:
;
; INVERTER: (inverter input output)
; AND-GATE: (and-gate input1 input2 output)
; OR-GATE:  (or-gate input1 input2 output)
; ============================================================================

;; Inverter
(define (inverter input output)
  (define (invert-input)
    (let ((new-value (logical-not (get-signal input))))
      (after-delay (lookup-variable-value 'inverter-delay the-global-environment)
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! input invert-input)
  'done)

(define (logical-not s)
  (cond ((= s 0) 1)
        ((= s 1) 0)
        (else (error "Invalid signal" s))))


;; AND-Gate
(define (and-gate a1 a2 output)
  (define (and-action-procedure)
    (let ((new-value
           (logical-and (get-signal a1) (get-signal a2))))
      (after-delay (lookup-variable-value 'and-gate-delay the-global-environment)
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 and-action-procedure)
  (add-action! a2 and-action-procedure)
  'done)


(define (logical-and x y)
  (if (and (= x 1) (= y 1))
      1
      0))



;; OR-Gate
```

```
(define (or-gate a1 a2 output)
  (define (or-action-procedure)
    (let ((new-value
           (logical-or (get-signal a1) (get-signal a2))))
      (after-delay (lookup-variable-value 'or-gate-delay the-global-environment)
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! a1 or-action-procedure)
  (add-action! a2 or-action-procedure)
  'done)


(define (logical-or x y)
  (if (or (= x 1) (= y 1))
      1
      0))



; ================================================================================
; Representing Wires as Objects
;
; A wire consists of two local state-variables:
; - a signal value
; - an amount of action-procedures that are evaluated if the
;   signal value changes
; ================================================================================

; Wire as an Object
(define (make-wire-obj)
  (let ((signal-value 0)
        (action-procedures '()))
    (define (set-my-signal! new-value)
      ; Tests if the signal-value has changed
      ; If so, calls each action-procedure with the auxiliary function call-each
      (if (not (= signal-value new-value))
          (begin (set! signal-value new-value)
                 (call-each action-procedures))
          'done))
    (define (accept-action-procedure proc)
      ; Adds the given procedure to the list of procedures to be called,
      ; after that, calls once the added procedure
      (set! action-procedures (cons proc action-procedures))
      (proc))
    (define (dispatch m)
      ; Verifies and calls the wanted procedure
      (cond ((equal? m 'get-signal) signal-value)
            ((equal? m 'set-signal!) set-my-signal!)
            ((equal? m 'add-action!) accept-action-procedure)
            (else (error "Unknown operation -- WIRE" m))))
    dispatch))


(define my-wire-queue '())

(define (make-wire)
  (let ((wire (make-wire-obj)))
    (set! my-wire-queue (cons wire my-wire-queue))
    wire
    ))

(define (reset-wire-queue list)
```

```
  (cond
    [(eq? list empty) (error "keine Listeneinträge vorhanden")]
    [(eq? (cdr list) empty) (set-signal! (car list) 0)]
    [else
     (begin
       (set-signal! (car list) 0)
       (reset-wire-queue (cdr list)))]))

#|
(define a (make-wire))
(define b (make-wire))
(set-signal! a 1)
(set-signal! b 1)
(get-signal a)
(reseturs my-wire-queue)
|#


;  Calls every element of a list consisting procedures without arguments
(define (call-each procedures)
  (if (null? procedures)
      'done
      (begin
        ((car procedures))
        (call-each (cdr procedures)))))

; Syntactic sugar for using normal procedural syntax to operate with local
; wire-operations
(define (get-signal wire)
  (wire 'get-signal))

(define (set-signal! wire new-value)
  ((wire 'set-signal!) new-value))

(define (add-action! wire action-procedure)
  ((wire 'add-action!) action-procedure))




; ============================================================================
; The Agenda
;
; The "Schedule" for the actions have to be done
; ============================================================================

(define (after-delay delay- action)
  (add-to-agenda! (+ delay- (current-time the-agenda))
                  action
                  the-agenda))

; Works on the agenda, evaluates all the agenda's procedures in a chronological
; manner. Propagate works as long on the agenda until there are elements in it.
(define (propagate)
  (if (empty-agenda? the-agenda)
      'done
      (let ((first-item (first-agenda-item the-agenda)))
        (first-item)
        (remove-first-agenda-item! the-agenda)
        (propagate))))
```

```
; ============================================================================
; A sample simulation
;
; To show the simulator in action
; ============================================================================

;; place a "probe" on a wire: whenever the signal on the wire changes its
;; value, it should print the new signal value, together with the current
;; time and a name that identifies the wire
(define (probe name wire)
  (add-action! wire (lambda ()
                      (let ((sTime (number->string (current-time the-agenda)))
                            (sSign (number->string (get-signal wire)))
                            (sName (symbol->string (my-eval name '()))))
                        (print-to-transcript (string-append sName
                                                            " - neuer Wert: "
                                                            sSign
                                                            ", aktuelle Zeit: "
                                                            sTime))
                        (transcript_prompt)))))


;; Display the current time
(define (get-current-time)

  (let ((sTime (number->string (current-time the-agenda))))
    (print-to-transcript (string-append "Aktuelle Zeit: "
                                        sTime))
    (transcript_prompt)
    ))


(define (simulation)
  (set! the-agenda (make-agenda))
  (set-variable-value! 'and-gate-delay and-gate-delay the-global-environment)
  (set-variable-value! 'or-gate-delay or-gate-delay the-global-environment)
  (set-variable-value! 'inverter-delay inverter-delay the-global-environment)
  (print-to-transcript "Simulation initialized! -- SIMULATION"))




; ============================================================================
; Implementing the agenda and its structure
;
; The agenda consists of time-segments. Every time-segment is a
; (cons time queue), so a time, where all the queue's procedures have to be
; started.
; In the agenda's head the current time is safed.
; ============================================================================



(define (make-time-segment time queue)
  (cons time queue))
(define (segment-time s) (car s))
(define (segment-queue s) (cdr s))

; a new agenda has no time-segments and a current-time of 0
(define (make-agenda) (list 0))
```

```
(define (current-time agenda) (car agenda))
(define (set-current-time! agenda time)
  (set-car! agenda time))

(define (segments agenda) (cdr agenda))
(define (set-segments! agenda segments)
  (set-cdr! agenda segments))
(define (first-segment agenda) (car (segments agenda)))
(define (rest-segments agenda) (cdr (segments agenda)))

(define (empty-agenda? agenda)
  (null? (segments agenda)))

; If the agenda is empty install the time-segment in the agenda.
; If not, scan the agenda chronologically for our appointed time and:
;   - the agenda is empty -> install a new time-segment
;   - the time is already in the agenda -> add our action in this queue
;   - matching a time after our time -> install directly
;     in front our time-segment
;   - if we pass the agenda's end -> install the time-segment at the end
(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments)
    (or (null? segments)
        (< time (segment-time (car segments)))))
  (define (make-new-time-segment time action)
    (let ((q (make-queue)))
      (insert-queue! q action)
      (make-time-segment time q)))
  (define (add-to-segments! segments)
    (if (= (segment-time (car segments)) time)
        (insert-queue! (segment-queue (car segments))
                       action)
        (let ((rest (cdr segments)))
          (if (belongs-before? rest)
              (set-cdr!
               segments
               (cons (make-new-time-segment time action)
                     (cdr segments)))
              (add-to-segments! rest)))))
  (let ((segments (segments agenda)))
    (if (belongs-before? segments)
        (set-segments!
         agenda
         (cons (make-new-time-segment time action)
               segments))
        (add-to-segments! segments))))

; removes the first item from the agenda
(define (remove-first-agenda-item! agenda)
  (let ((q (segment-queue (first-segment agenda))))
    (delete-queue! q)
    (if (empty-queue? q)
        (set-segments! agenda (rest-segments agenda)))))

; whenever we extract an item, we also have to update the current time
(define (first-agenda-item agenda)
  (if (empty-agenda? agenda)
      (error "Agenda is empty -- FIRST-AGENDA-ITEM")
      (let ((first-seg (first-segment agenda)))
        (set-current-time! agenda (segment-time first-seg))
        (front-queue (segment-queue first-seg)))))
```

```
;; DEFINITION OF THE-AGENDA
(define the-agenda (make-agenda))
```

A.6 Datei "Queues.scm"

```
; ____________________________________________________________________________
; PROJEKT INFORMATIK, 2004/05
; UNIVERSITAET FREIBURG_______________________________________________________


; Autor:    Urs Sieber
; Datum:    24.03.2005
; Filename: Queues.scm




; ************** SHORT DESCRIPTION OF THE PURPOSE OF THIS FILE ****************

; The code is taken from the following book:

;   - Book    : Structure and Interpretetion of Computer Programs
;   - Edition : Second Edition
;   - Autor   : Harold Abelson & Gerald Jay Sussman
;   - Section : 3.3.2 Representing Queues
;   - Pages   : 261 to 265

; ******************************************************************************




; ############################################################################
; SECTION 3.3.2 Representing Queues
; ############################################################################


; Following procedures to select and modify the queues front and rear pointer
(define (front-ptr queue) (car queue))
(define (rear-ptr queue) (cdr queue))
(define (set-front-ptr! queue item) (set-car! queue item))
(define (set-rear-ptr! queue item) (set-cdr! queue item))


; ============================================================================
; QUEUES:
; A queue is a sequence of elemets, where at the beginning elements are deleted
; and at the end elemets are added. The following procedures are used
; to operate with queues.
; ============================================================================

; Constructor:
; An emtpy queue consists of a cons and two empty lists as elements.
(define (make-queue) (cons '() '()))

; Selectors that test if a element is a front-element of the queue and that
; test the empty queue
(define (empty-queue? queue) (null? (front-ptr queue)))
(define (front-queue queue)
```

```
  (if (empty-queue? queue)
    (error 'front-queue "FRONT called with an empty queue " queue)
    (car (front-ptr queue))))

; Inserts a new element and sets the pointers to the first and the last element
; of the queue
(define (insert-queue! queue item)
  (let ((new-pair (cons item '())))
    (cond ((empty-queue? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair)
           queue)
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)
           queue))))

; "Delets" or hides the queues' first element by setting the front pointer to
; the cdr ("2nd element") of the queue.
(define (delete-queue! queue)
  (cond ((empty-queue? queue)
         (error 'delete-queue! "DELETE! called with an empty queue " queue))
        (else
         (set-front-ptr! queue (cdr (front-ptr queue)))
         queue)))
```

A.7 Datei "ReadEvalPrint.scm"

```
; ------------------------------------------------------------------------
; PROJEKT PROGRAMMIERUNG, 2004/05
; UNIVERSITAET FREIBURG------------------------------------------------


; Autor:    Lehmann Norbert + Christian Eichenberger + Cesar Schneuwly
; Datum:    16.12.2004
; Filename: ReadEvalPrint.scm


; ****************************************************************
; In oder to use this file together with the Metacircular Evaluator
; presented in the book of Abelson "Structure and Interpretation of
; Computer Programs", the functions "my-eval" & "eval-sequence" and
; the definition of the global variable "the-global-environment"
; have to be changed.
; It is also supposed that "eval" and "apply" of the metacircular
; evaluator were renamed to "my-eval" and "my-apply".
;
; The Transcript Window is opened by evaluating the command
; "(setup-transcript)" (see the last line of this file).
; ****************************************************************

; loading the PLT Framework libraries...
(require (lib "framework.ss" "framework"))

; ************************************************************************
; DESCRIPTION: These should of course be changed...

#|
(define (my-eval exp env)
  exp)
```

```
(define (eval-sequence exp env)
  (print-to-transcript (list-to-string exp))
  (transcript_prompt))
|#

; these variables are used for the transcript...
(define TranscriptWindow #f)
(define TranscriptEditor #f)



; *************************************************************************
; DESCRIPTION: Some fonctions for writing into the transcript...

(define (transcript_welcome)
  (send TranscriptEditor insert "Welcome, insert your commands please ..."))

(define (transcript_prompt)
  (send TranscriptEditor set-position (send TranscriptEditor last-position))
  (send TranscriptEditor insert #\newline)
  (send TranscriptEditor insert ">> "))


(define (print-error-msg theMessage)
  (print-to-transcript (string-append "Error: " theMessage))
  (transcript_prompt))


(define (print-to-transcript TheThing)
  (let ((NewString (ConvertToString TheThing)))
    (send TranscriptEditor set-position
          (send TranscriptEditor last-position))
    (send TranscriptEditor insert NewString)))



; *************************************************************************
; DESCRIPTION: Functions for converting strings and lists...

(define (ConvertToString TheThing)
  (cond ((string? TheThing) (string-append "\"" TheThing "\""))
        ((number? TheThing) (number->string TheThing))
        ((symbol? TheThing) (symbol->string TheThing))
        ((primitive-procedure? TheThing) "#<PRIM-PROC>")
        ((compound-procedure? TheThing) "#<COMPOUND-PROC>")
        ((list? TheThing) (list-to-string TheThing))
        ((pair? TheThing) (string-append "("
                                         (ConvertToString (car TheThing))
                                         " . "
                                         (ConvertToString (cdr TheThing))
                                         ")"))
        ((eq? TheThing #f) "#f")
        ((eq? theThing #t) "#t")
        (else "")))


; Converts a list into a string
; (attention, apply must be the "apply" from DrScheme)...
(define (list-to-string theList)
  (if (null? theList)
      "()"
```

```
      (string-append "("
                     (ConvertToString (car theList))
                     (apply string-append
                            (map (lambda (item)
                                   (string-append " " (ConvertToString item)))
                                 (cdr theList)))
                     ")")))


; Convert a string into a list...
(define (extended-read string)
  (read-from-string-all string (lambda(msg) (print-error-msg msg))))



; ************************************************************************
; DESCRIPTION: The driver-loop -> "read-eval-print" loop

(define (driver-loop)
  (with-handlers ([exn? (lambda (exn) (print-error-msg (exn-message exn)))])
    (let* ((EndPos (send TranscriptEditor get-end-position))
           (StartPos (find-start-position EndPos)))
 (if StartPos
  (if (scheme:text-balanced? TranscriptEditor StartPos EndPos)
    (let* ((theText (send TranscriptEditor get-text StartPos EndPos #f #f))
           (TheSequence (extended-read theText))
           ;  old Version
           ; (TheResult (my-eval (car TheSequence) the-global-environment)))
           ;  If there is an empty sequence, the if prints a control message
             (TheResult (if (empty? TheSequence)
               "Empty Input - insert your commands here..."
               (my-eval (car TheSequence) the-global-environment))))
             (print-to-transcript TheResult)
             (transcript_prompt))
             (send TranscriptEditor insert #\newline))
             (transcript_prompt)))))



; ************************************************************************
; DESCRIPTION: Searchs the current expression. The result is the starting
;              position. We will search the character #\> preceded by
;              the character #\newline.

(define (find-start-position EndPos)
  (define (iterSearch pos)
    (cond ((< pos 0) #f)
          ((and (char=? (send TranscriptEditor get-character pos) #\>)
                (char=? (send TranscriptEditor get-character (- pos 1))
                        #\newline))
           (+ pos 2))
          (else (iterSearch (- pos 1)))))
  (IterSearch (- EndPos 1)))



; ************************************************************************
; DESCRIPTION: The editor class for the transcript...

;(define transcript:return%
;  (class text:return% (mode:host-text<%>) (init (return driver-loop))
;    (super-instantiate () (return return))))
```

```
(define transcript:editor%
   (class scheme:text%
     (rename  (super-on-local-char on-local-char))
     (override* (on-local-char
                 (lambda (evt) (super-on-local-char evt)
                               (if (eq? (send evt get-key-code) #\return)
                                   (driver-loop)))))
     (super-instantiate ())
     ))



; ************************************************************************
; DESCRIPTION: The frame classes used for the transcript and the editor...

(define my-frame%
  (class frame:searchable% (init label width height x y style)
    (override*
     (file-menu:new-callback (lambda (x y) (my-new-proc x y)))
     (file-menu:open-callback (lambda (x y) (my-open-proc x y)))
     (file-menu:save-as-callback (lambda (x y) (my-save-proc x y)))
     (help-menu:create-about? (lambda () #t))
     (help-menu:about-string (lambda () "About DigSim..."))
     (help-menu:about-callback (lambda (x y) (show-about-dialog)))
     (edit-menu:create-preferences? (lambda () #f)))
    (super-instantiate
        (label) (width width) (height height) (x x) (y y) (style style))))

(define frame:transcript%
  (class my-frame% (init label width height x y style)
    (override*
     (get-editor<%> (lambda () scheme:text<%>))
     (get-editor% (lambda () transcript:editor%)))
    (super-instantiate (label width height x y style))))

(define frame:scheme-editor%
  (class my-frame% (init label width height x y style)
    (override*
     (get-editor<%> (lambda () scheme:text<%>))
     (get-editor% (lambda () scheme:text%)))
    (super-instantiate (label width height x y style))))



; ************************************************************************
; DESCRIPTION: Creating of the transcript window...

(define (setup-transcript)
  (set! TranscriptWindow
        (instantiate frame:transcript% ("Transcript" 500 300 4 400 '())))
  (set! TranscriptEditor (send TranscriptWindow get-editor))
  (send TranscriptEditor erase)
  (transcript_welcome)
  (transcript_prompt)
  (add-menus TranscriptWindow)
  (send TranscriptWindow show #t))



; ************************************************************************
; DESCRIPTION: Functions for the menus...
```

```
(define (my-open-proc item control)
  (let ((theFile (finder:get-file)))
    (if theFile
        (let ((theWindow (make-object frame:scheme-editor%
                           (file-name-from-path theFile) 500 300 4 10 '())))
          (send (send theWindow get-editor) load-file theFile 'text #f)
          (add-menus theWindow)
          (send theWindow show #t)))))

(define (my-new-proc item control)
  (let ((theWindow (make-object frame:scheme-editor%
                     (gui-utils:next-untitled-name) 500 300 4 10 '())))
    (add-menus theWindow)
    (send theWindow show #t)))

(define (my-save-proc item control)
  (let ((theWindow (send (send (send item get-parent) get-parent) get-frame))
        (theFile (finder:put-file)))
    (when theFile
      (send (send theWindow get-editor) save-file theFile 'text #f)
      (send theWindow set-label (file-name-from-path theFile)))))



; ************************************************************************
; DESCRIPTION: Starting up...

;(setup-transcript)
```

A.8 Datei "MetaCircularEvaluator.scm"

```
; ------------------------------------------------------------------------------
; PROGRAMMIERUNG: PROJEKT 2004/2005
; UNIVERSITAET FREIBURG_____________________________________________________


; Autoren:  Urs Sieber
; Datum:    24.01.2005
; Filename: MetaCircularEvaluator.scm



; ************** SHORT DESCRIPTION OF THE PURPOSE OF THIS FILE ***************
;
; Some routines for implementing the metacircular evaluator of Scheme.
; The code is taken from the following book:
;
;   - Book    : Structure and Interpretetion of Computer Programs
;   - Edition : Second Edition
;   - Autor   : Harold Abelson & Gerald Jay Sussman
;   - Section : 4.1.1 Operations on Environments
;   - Pages   : 364 to 384
;
; ****************************************************************************


; ############################################################################
; SECTION 4.1.1 The Core of the Evaluator:
;
```

```
; The process of evaluation can be seen as the interplaying procedures:
; EVAL and APPLY
; ###############################################################################

; ===============================================================================
; EVAL:
; Eval classifies the expression and rules its evaluation.
; Eval is taken as a cond differentiation for each syntactic expression-type.
; ===============================================================================

(define (my-eval exp env)
 (cond
  ;; DigSim
  ((and-gate? exp)         (eval-and-gate exp env))          ;DigSim-extension
  ((or-gate? exp)          (eval-or-gate exp env))           ;DigSim-extension
  ((inverter? exp)         (eval-inverter exp env))          ;DigSim-extension
  ((simulation? exp)       (eval-simulation exp env))        ;DigSim-extension
  ((propagate? exp)        (eval-propagate exp env))         ;DigSim-extension
  ((probe? exp)            (eval-probe exp env))             ;DigSim-extension
  ((get-signal? exp)       (eval-get-signal exp env))        ;DigSim-extension
  ((set-signal? exp)       (eval-set-signal exp env))        ;DigSim-extension
  ((make-wire? exp)        (eval-make-wire exp env))         ;DigSim-extension
  ((wires? exp)            (eval-wires exp env))             ;DigSim-extension
  ((inst? exp)             (eval-inst exp env))              ;DigSim-extension
  ((defmodul? exp)         (eval-defmodul exp env))          ;DigSim-extension
  ((intern? exp)           (eval-intern exp env))            ;DigSim-extension
  ((get-current-time? exp) (eval-get-current-time exp env))  ;DigSim-extension
  ((display? exp)          (my-display                       ;DigSim-extension
                             (my-eval (cadr exp) env)))

  ;; Scheme
  ((self-evaluating? exp) exp)
  ((variable? exp)         (lookup-variable-value exp env))
  ((quoted? exp)           (text-of-quotation exp))
  ((assignment? exp)       (eval-assignment exp env))
  ((definition? exp)       (eval-definition exp env))
  ((and? exp)              (eval-and exp env))
  ((or? exp)               (eval-or exp env))
  ((if? exp)               (eval-if exp env))
  ((lambda? exp)           (make-procedure (lambda-parameters exp)
                             (lambda-body exp)
                                env))
  ((begin? exp)            (eval-sequence (begin-actions exp) env))
  ((let? exp)              (eval-let exp env))
  ((let*? exp)             (eval-let* exp env))
  ((cond? exp)             (my-eval (cond->if exp) env))
  ((application? exp)      (my-apply (my-eval (operator exp) env)
                             (list-of-values (operands exp) env)))
  (else
   (error "Unknown expression type -- EVAL" exp))
 )
)



; ***************************************************************************
; DESCRIPTION: The Special Form AND

(define (and? exp)
  (tagged-list? exp 'and))

(define (eval-and exp env)
```

```
  (define (iter arguments)
    (if (null? (car arguments)) #t
        (if (NOT (my-eval (car arguments) env)) #f
            (iter (cdr exp) env))))
  (iter (cdr exp)))
```

```
; *************************************************************************
; DESCRIPTION: The Special Form OR

(define (or? exp)
  (tagged-list? exp 'or))

(define (eval-or exp env)
  (define (iter arguments)
    (if (null? (car arguments)) #f
        (if (my-eval (car arguments) env) #t
            (iter (cdr exp) env))))
  (iter (cdr exp) env))
```

```
; *************************************************************************
; DESCRIPTION: The Special Form LET
; A derived expression from lambda.

(define (let? exp)
  (tagged-list? exp 'let))

(define (eval-let exp env)
  (my-eval (let->lambda exp) env))

; Reorganisates a let-expression in a lambda expression.
; The parameters are found with auxiliary functions.
(define (let->lambda exp)
 (cons
  (make-lambda (let-vars exp)
               (let-body exp))
  (let-vals exp))
 )


; let-body: exp -> list
(define (let-body exp)
  (cddr exp))

; Generates a list of the variables out of the single let-expressions.
; Variables are safed in (car (cadr exp)).
; let-vars: exp -> list-of-variables
(define (let-vars exp)
 (define (let-var zuweisungen)
  (cond
    [(null? (cdr zuweisungen)) (cons (caar zuweisungen) empty)]
    [else
     (cons (caar zuweisungen) (let-var (cdr zuweisungen)))]))
  (let-var (cadr exp)))
```

```
; Difference to let-vars: Values are safed in (cadr (cadr exp)).
; let-vals: exp -> list-of-values
(define (let-vals exp)
 (define (let-val zuweisungen)
  (cond
    [(null? (cdr zuweisungen)) (cons (cadar zuweisungen) empty)]
    [else
     (cons (cadar zuweisungen) (let-val (cdr zuweisungen)))]))
  (let-val (cadr exp)))




; **************************************************************************
; DESCRIPTION: The Special Form LET*
; A derived expression from let.

(define (let*? exp)
  (tagged-list? exp 'let*))

; evaluates the enclosed (geschachtelten) let-expressions.
(define (eval-let* exp env)
  (my-eval (let*->geschachtelte-lets exp) env))

; generates an enclosed (geschachteltes) let out of let*, which body
; contents newly the let with the next setting (Zuweisung).
; If there is just one setting, the normal evaluation with let is suffisant.
; The auxiliary function next-let outputs the following enclosed
; (geschachtelte) let.
(define (let*->geschachtelte-lets exp)
    (if (null? (cdadr exp))
        (cons 'let (cdr exp))
        (list 'let (list
                    (list (car (let-vars exp))
                          (car (let-vals exp))))
                    (next-let* exp))))


; next-let* generates a new let* with all the following settings
;(Zuweisungen) without the first.
(define (next-let* exp)
  (let*->geschachtelte-lets
   (cons (car exp) (cons (cdadr exp) (cddr exp)))))




; ============================================================================
; APPLY
; Apply applies a procedure on a list of arguments.
; It classifies two major procedures:
; It calls apply-primitive-procedure on primitve procedures.
; For compound procedures expressions in the body of the procedure are applied
; sequential.
; ============================================================================

(define (my-apply procedure arguments)
  (cond ((primitive-procedure? procedure)
```

```
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment
           (procedure-parameters procedure)
           arguments
           (procedure-environment procedure))))
        (else
         (error
          "Unknown procedure type -- APPLY " procedure))))



; ========================================================================
; Procedure arguments
; ========================================================================

(define (list-of-values exps env)
  (if (no-operands? exps)
      '()
      (cons (my-eval (first-operand exps) env)
            (list-of-values (rest-operands exps) env))))




; ========================================================================
; Conditionals
; ========================================================================

(define (eval-if exp env)
  (if (true? (my-eval (if-predicate exp) env))
      (my-eval (if-consequent exp) env)
      (my-eval (if-alternative exp) env)))




; ========================================================================
; Sequences
; ========================================================================

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (my-eval (first-exp exps) env))
        (else (my-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))




; ========================================================================
; Assignments and Definitions
; ========================================================================

; eval-assignment deals with assignments on variables.
; set-variable-value <var> <value> <env>: installs a new variable in env and
; changes the variable's binding in the env environment that the variable is
; now bound at the val value.
(define (eval-assignment exp env)
```

```
  (set-variable-value! (assignment-variable exp)
                       (my-eval (assignment-value exp) env)
                       env)
  'done)

; Handling of definitions
; define-variable! <var> <value> <env>: adds the first frame in env
; the binding <var> <val>.
(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (my-eval (definition-value exp) env)
                    env)
  'done)




; ============================================================================
; DigSim-Extensions
; ============================================================================

;; and-gate
(define (and-gate? exp)
  (tagged-list? exp 'and-gate))

(define (eval-and-gate exp env)       ;; Syntax Tests
  (if (not (empty? (cdr exp)))
      (if (not (empty? (cddr exp)))
          (if (not (empty? (cdddr exp)))
              (and-gate (lookup-variable-value (cadr exp) env)
                        (lookup-variable-value (caddr exp) env)
                        (lookup-variable-value (cadddr exp) env))
              (error 'eval-and-gate "Wrong Input!"))
          (error 'eval-and-gate "Wrong Input!"))
      (error 'eval-and-gate "Wrong Input!")))

;; or-gate
(define (or-gate? exp)
  (tagged-list? exp 'or-gate))

(define (eval-or-gate exp env)
  (if (not (empty? (cdr exp)))        ;; Syntax Tests
      (if (not (empty? (cddr exp)))
          (if (not (empty? (cdddr exp)))
              (or-gate (lookup-variable-value (cadr exp) env)
                       (lookup-variable-value (caddr exp) env)
                       (lookup-variable-value (cadddr exp) env))
              (error 'eval-or-gate "Wrong Input!"))
          (error 'eval-or-gate "Wrong Input!"))
      (error 'eval-or-gate "Wrong Input!")))

;; inverter
(define (inverter? exp)
  (tagged-list? exp 'inverter))

(define (eval-inverter exp env)
  (if (not (empty? (cdr exp)))        ;; Syntax Tests
      (if (not (empty? (cddr exp)))
          (inverter (lookup-variable-value (cadr exp) env)
                    (lookup-variable-value (caddr exp) env))
          (error 'eval-inverter "Wrong Input!"))
      (error 'eval-inverter "Wrong Input!")))
```

```
;; simulation
(define (simulation? exp)
  (tagged-list? exp 'simulation))

(define (eval-simulation exp env)
  (simulation))

;; propagate
(define (propagate? exp)
  (tagged-list? exp 'propagate))

(define (eval-propagate exp env)
  (propagate))

;; probe
(define (probe? exp)
  (tagged-list? exp 'probe))

(define (eval-probe exp env)
  (if (not (empty? (cdr exp)))
      (if (not (empty? (cddr exp)))
          (probe (cadr exp) (lookup-variable-value (caddr exp) env))
          (error 'eval-probe "Wrong Input!"))
      (error 'eval-probe "Wrong Input!")))

;; get-signal
(define (get-signal? exp)
  (tagged-list? exp 'get-signal))

(define (eval-get-signal exp env)
  (if (not (empty? (cdr exp)))
      (get-signal (lookup-variable-value (cadr exp) env))
      (error 'eval-get-signal "Wrong Input!")))

;; set-signal!
(define (set-signal? exp)
  (tagged-list? exp 'set-signal!))

(define (eval-set-signal exp env)
  (if (not (empty? (cdr exp)))
      (if (not (empty? (cddr exp)))
          (set-signal! (lookup-variable-value (cadr exp) env) (my-eval (caddr exp) env))
          (error 'eval-set-signal "Wrong Input!"))
      (error 'eval-set-signal "Wrong Input!")))

;; make-wire
(define (make-wire? exp)
  (tagged-list? exp 'make-wire))

(define (eval-make-wire exp env)
  (make-wire))

;; wires
(define (wires? exp)
  (tagged-list? exp 'wires))

;; checks syntax
(define (eval-wires exp env)
  (if (empty? (cddr exp))
      (install-wire (cadr exp) env)
      (error 'eval-wires "Wrong Input!")))
```

```
;; assigns the variables the function make-wire and defines the variables.
(define (install-wire exp env)
  (cond
    [(empty? exp) 'done]
    [else (define-variable! (car exp) (make-wire) env)
          (install-wire (cdr exp) env)]))

;; inst
(define (inst? exp)
  (tagged-list? exp 'inst))

;; eval-inst is Syntactic Sugar:
;; Instead of (x-gate i1 i2 o1) (inst x-gate (i1 i2) (o1)) is allowed.
;; Inputs and Outputs are apart as arguments.
;; The function changes the inst expression in a "normal"
;; x-gate expression and evaluates it as defined.
(define (eval-inst exp env)
  (my-eval (append (list (cadr exp))           ;; 'Symbol
                   (caddr exp)                 ;; Inputs
                   (cadddr exp))               ;; Outputs
           env))


;; defmodul
(define (defmodul? exp)
  (tagged-list? exp 'defmodul))

;; eval-defmodule is Syntactic Sugar:
;; Instead of (define (<module-name> <input intern output>) <body>)
;; (defmodul <module-name> (<input intern>) (<output>) <body>) is allowed.
;; The function changes the defmodul expression in a define and evaluates it.
(define (eval-defmodul exp env)
  (let ((module-name (cadr exp))               ;; 'Module name
        (inputs (caddr exp))                   ;; Input-, Intern wires
        (outputs (cadddr exp))                 ;; Outputs
        (body (cddddr exp)))                   ;; Body
    (my-eval (append (list 'define)
                     (list (append (list module-name)
                                   inputs
                                   outputs))
                     body)
             env)))


;; intern
(define (intern? exp)
  (tagged-list? exp 'intern))

;; eval-intern is Syntactic Sugar:
;; Instead of (let ((i1 (make-wire))...) <body>) (intern (i1 i2 i3...)
;; <body>) is allowed.
;; The function changes the intern in a lambda expression.
(define (eval-intern exp env)
  (let ((wires (get-intern-wires (cadr exp)))
        (body (cddr exp)))
    (my-eval (cons (make-lambda '() (append wires body)) empty) env)))

(define (get-intern-wires exp)
  (if (empty? exp)
      '()
      (append (cons (append (cons 'define empty)
```

```
                            (cons (car exp) empty)
  ;; produces a list of (define wire (make-wire)) as input parameter in lambda.
                            (cons (cons 'make-wire empty) empty)) empty)
              (get-intern-wires (cdr exp)))))

;; get-current-time
(define (get-current-time? exp)
  (tagged-list? exp 'get-current-time))

(define (eval-get-current-time exp env)
  (get-current-time))

;; Display
(define (display? exp)
  (tagged-list? exp 'display))

(define (my-display theThing)
  (send TranscriptEditor insert #\newline)
  (send TranscriptEditor insert (ConvertToString theThing))
  (send TranscriptEditor insert #\newline))


; ###############################################################################
; SECTION 4.1.2 Representing Expressions
; ###############################################################################

;; The only self-evaluating items are numbers and strings.
(define (self-evaluating? exp)
  (cond ((number? exp) #t)
        ((string? exp) #t)
        ((eq? exp #t) #t)
        ((eq? exp #f) #t)
        (else #f)))

;; Quotations have the form (quote <text-of-quotation>)
(define (quoted? exp)
  (tagged-list? exp 'quote))

;; Returns the text of a quotation
(define (text-of-quotation exp) (cadr exp))

;; Tagged-list? identifies list beginning with a designated symbol.
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      #f))

;; Variables are represented by symbols
(define (variable? exp) (symbol? exp))

;; Assignments have the form (set! <var> <value>)
(define (assignment? exp)
  (tagged-list? exp 'set!))

(define (assignment-variable exp) (cadr exp))

(define (assignment-value exp) (caddr exp))

;; Definitions have the form (define <var> <value>)
;; or (define (<var> <parameter_1> ... <parameter_n>) <body>)
(define (definition? exp)
  (tagged-list? exp 'define))
```

```
(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp)
      (caadr exp)))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cdadr exp)
                   (cddr exp))))

;; Lambda expressions are lists that begin with the symbol lambda.
(define (lambda? exp) (tagged-list? exp 'lambda))

(define (lambda-parameters exp) (cadr exp))
(define (lambda-body exp) (cddr exp))

(define (make-lambda parameters body)
  (cons 'lambda (cons parameters body)))

;; Conditionals begin with if and have a predicate, a consequent,
;; and an (optional) alternative.
;; If the expressions has no alternative part,
;; we provide false as the alternative.
(define (if? exp) (tagged-list? exp 'if))

(define (if-predicate exp) (cadr exp))

(define (if-consequent exp) (caddr exp))

(define (if-alternative exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      #f))

(define (make-if predicate consequent alternative)
  (list 'if predicate consequent alternative))

;; Begin packages a sequence of expressions into a single expression.
;; We include syntax operations on begin expressions to extract the actual
;; sequence from the begin expression, as well as selectors that return
;; the first expression and the rest of the expression in the sequence.
(define (begin? exp) (tagged-list? exp 'begin))

(define (begin-actions exp) (cdr exp))

(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))

(define (sequence->exp seq)
  (cond ((null? seq) seq)
        ((last-exp? seq) (first-exp seq))
        (else (make-begin seq))))

(define (make-begin seq) (cons 'begin seq))

;; A procedure application is any compound expression that is not one
;; of the above expression types. The car of the expression is the
;; operator, and the cdr is the list of operands:
(define (application? exp) (pair? exp))
```

```
(define (operator exp) (car exp))
(define (operands exp) (cdr exp))

(define (no-operands? ops) (null? ops))
(define (first-operand ops) (car ops))
(define (rest-operands ops) (cdr ops))
```

```
; ============================================================================
; Derived Expressions COND:
; Cond can be implemented as a nest of if expressions. We include syntax
; procedures that extract the parts of a cond expression, and a procedure
; cond->if that transforms cond expressions into if expressions.
;
;   - A case analysis begins with cond and has a list of predicate-action
;     clauses.
;   - A clause is an else clause if its predicate is the symbol else.
; ============================================================================

(define (cond? exp) (tagged-list? exp 'cond))

(define (cond-clauses exp) (cdr exp))

(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))

(define (cond-predicate clause) (car clause))

(define (cond-actions clause) (cdr clause))

(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))

(define (expand-clauses clauses)
  (if (null? clauses)
      #f                          ; no else clause
      (let ((first (car clauses))
            (rest (cdr clauses)))
        (if (cond-else-clause? first)
            (if (null? rest)
                (sequence->exp (cond-actions first))
                (error "ELSE clause isn't last -- COND->IF"
                       clauses))
            (make-if (cond-predicate first)
                     (sequence->exp (cond-actions first))
                     (expand-clauses rest))))))
```

```
; ############################################################################
; SECTION 4.1.3 Evaluator Data Structures
; ############################################################################

; ============================================================================
; Testing of predicates
; ============================================================================
```

```
(define (true? x)
  (not (eq? x #f)))

(define (false? x)
  (eq? x #f))



; ======================================================================
; Representing Procedures
; ======================================================================

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))

(define (compound-procedure? p)
  (tagged-list? p 'procedure))


(define (procedure-parameters p) (cadr p))
(define (procedure-body p) (caddr p))
(define (procedure-environment p) (cadddr p))




; ##############################################################################
; SECTION 4.1.4 Running the Evaluator as a Program
; ##############################################################################

(define (setup-environment)
  (let ((initial-env
         (extend-environment (primitive-procedure-names)
                             (primitive-procedure-objects)
                             the-empty-environment)))
    ;; Scheme globals have also to be defined
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    ;; Global variables in DigSim
    (define-variable! 'inverter-delay inverter-delay initial-env)
    (define-variable! 'and-gate-delay and-gate-delay initial-env)
    (define-variable! 'or-gate-delay or-gate-delay initial-env)
    initial-env))

;; Tests whether <proc> is a primitive procedure.
(define (primitive-procedure? proc)
  (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))

;; List of primitve procedures
(define primitive-procedures
  `((car ,car)
    (cdr ,cdr)
    (caar ,caar)
    (cadr ,cadr)
    (cdar ,cdar)
    (cddr ,cddr)
    (caaar ,caaar)
    (caadr ,caadr)
    (cadar ,cadar)
```

```
(cdaar ,cdaar)
(caddr ,caddr)
(cddar ,cddar)
(cdadr ,cdadr)
(cdddr ,cdddr)
(caaaar ,caaaar)
(caaadr ,caaadr)
(caadar ,caadar)
(cadaar ,cadaar)
(cdaaar ,cdaaar)
(caaddr ,caaddr)
(caddar ,caddar)
(cddaar ,cddaar)
(cdaadr ,cdaadr)
(cadadr ,cadadr)
(cdadar ,cdadar)
(cdddar ,cdddar)
(cadddr ,cadddr)
(cddddr ,cddddr)
(cons ,cons)
(null? ,null?)
(pair? ,pair?)
(set-car! ,set-car!)
(set-cdr! ,set-cdr!)
(list ,list)
(length ,length)
(list-tail ,list-tail)
(list-ref ,list-ref)
(append ,append)
(memq ,memq)
(memv ,memv)
(member ,member)
(assq ,assq)
(assv ,assv)
(assoc ,assoc)
(not ,not)
(= ,=)
(< ,<)
(> ,>)
(<= ,<=)
(>= ,>=)
(+ ,+)
(- ,-)
(* ,*)
(/ ,/)
(remainder ,remainder)
(modulo ,modulo)
(exp ,exp)
(expt ,expt)
(random ,random)
(eq? ,eq?)
(eqv? ,eqv?)
(equal? ,equal?)
(sin ,sin)
(cos ,cos)
(tan ,tan)
(asin ,asin)
(acos ,acos)
(atan ,atan)
(abs ,abs)
(not ,not)
(string->list ,string->list)
```

```
    (string->number ,string->number)
    (string->symbol ,string->symbol)
    (string-append ,string-append)
    (display ,display)
    (and-gate ,and-gate)              ;; Dig-Sim primitive, as in Simulator.scm
    (or-gate ,or-gate)                ;; Dig-Sim primitive, as in Simulator.scm
    (inverter ,inverter)              ;; Dig-Sim primitive, as in Simulator.scm
    (make-wire ,make-wire)            ;; Dig-Sim primitive, as in Simulator.scm
    (simulation ,simulation)          ;; Dig-Sim primitive, as in Simulator.scm
    (propagate ,propagate)            ;; Dig-Sim primitive, as in Simulator.scm
    (get-signal ,get-signal)          ;; Dig-Sim primitive, as in Simulator.scm
    (set-signal! ,set-signal!)        ;; Dig-Sim primitive, as in Simulator.scm
    (probe ,probe)                    ;; Dig-Sim primitive, as in Simulator.scm
    ))
; ============================================================================
(define (primitive-procedure-names)
  (map car
       primitive-procedures))

(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc)))
       primitive-procedures))

(define (apply-primitive-procedure proc args)
  (apply (primitive-implementation proc) args))

(define input-prompt "=> ")
(define output-prompt "== ")

#|
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (my-eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))
|#

(define (prompt-for-input string)
  (newline) (newline) (display string))

(define (announce-output string)
  (newline) (display string))

(define (user-print object)
  (if (compound-procedure? object)
      (display (list 'compound-procedure
                     (procedure-parameters object)
                     (procedure-body object)
                     '<procedure-env>))
      (display object)))


;; The Global Environment
(define the-global-environment (setup-environment))
'METACIRCULAR-EVALUATOR-LOADED

; uncomment for using the driver-loop,
; i.e. testing the evaluator interactively
;(driver-loop)
```

A.9 Datei “standardMain.scm”

```
; ------------------------------------------------------------------------
; PROJEKT INFORMATIK, 2004/05
; UNIVERSITAET FREIBURG-------------------------------------------------


; Autor:    Urs Sieber
; Datum:    22.03.2005
; Filename: standardMain.scm

(require (lib "framework.ss" "framework"))

; the global variables...

  (define inverter-delay 2)
  (define and-gate-delay 3)
  (define or-gate-delay 5)


; loading the corresponding files in an appropriate manner...

  ; Starting Window
  (load "myAboutDialog.scm")

  ; New Menue Items
  (load "myMenus.scm")

  ; Settings for the delays / Delay Window
  (load "DelayDialog.scm")

  ; Implementing of an environment model
  (load "EnvironmentModel.scm")

  ; Specifications for a simulator for digital circuits
  ; DigSim extensions
  (load "Simulator.scm")

  ; Procedures for a sequence of elements, needed for the agenda
  (load "Queues.scm")

  ; New Interface / Transcript Window
  (load "ReadEvalPrint.scm")

  ; Metacircular Evaluator (main file)
  (load "MetaCircularEvaluator.scm")


(show-about-dialog)
(setup-transcript)
```

B CD mit DigSim und Dokumentation

Die unten beinhaltete CD sollte beim Einschieben in ein CD-Rom Laufwerk automatisch das Programm DigSim mittels einer `autorun.inf` Datei starten.

Ist dies nicht der Fall, so startet man das Programm manuell, indem man im Ordner `executable` das Programm `DigSim.exe` mit Doppelklick startet.

Auf unten angefügter CD sind folgende Daten gespeichert:

- DigSim stand-alone Applikation im Ordner `executable`
- Source Files des gesamten Projekts im Ordner `source`
- Dokumentation interaktiv (mit Hyperreferenz-Links) und Tex-File im Ordner `documentation`
- Test-Files für die Probe und Simulation von DigSim im Ordner `tests`