Facharbeit

Implementation eines Turingsimulators

Jan Engelhardt

17. März 2005

Max-Plank-Gymnasium Göttingen
Leistungskurs Informatik
Stufe 12 / Abiturjahrgang 2006

Kursleiter:	Dr. Eckart Modrow
Kursthema:	Informatik II
Bearbeitungszeit:	03. Februar 2005–17. März 2005
Abgabetermin:	Donnerstag, 17. März 2005

Inhaltsverzeichnis

1 Einleitung

1.1 Grenzen gängiger Turingsimulatoren

Für Turingsimulatoren gibt es eine Reihe von häufig verwendeten Methoden, um Zustandstabellen, auch Automatentafeln genannt, zu definieren. Dazu gehören statisch in das ausführbare Programm (Executable) einkompilierte Arrays[1], Strukturen oder Funktionen.

Der größte Nachteil bei diesen Methoden ist, dass bei Änderungen Teile des Quellcodes neu kompiliert werden müssen, was üblicherweise die Installation eines Compiler (u.U. auch einer ganzen Entwicklungsumgebung) erfordert, sofern dies noch nicht geschehen ist[2]. Steht der Quellcode gar nicht erst zur Verfügung, sind die Grenzen schon erreicht.

Um letztgenanntes Problem zu umgehen, können kompilierte Strukturen / Funktionen dynamisch nachgeladen werden. Abgesehen davon, dass nicht alle Programmiersprachen Shared Libraries[3] unterstützen, wird auch hier immer noch ein Compiler (oder Interpreter) benötigt.

Diese beiden Ansätze entsprechen den Ansprüchen für z.B. kurzlebige Demonstrationsprogramme — wer mehr Funktionalität braucht, wird an das Internet verwiesen. Solange Programme mit fixen Zustandstabellen nicht die Oberhand im Internet ergreifen, besitzen die meisten grafischen Turingsimulatoren zumindest eine Tabelle oder andere Eingabemethoden, in der man die Operationen definieren kann.

Doch auch solche tabellenbasierte Simulatoren sind nicht endlos skalierbar: mit ansteigender Anzahl der Zustände wird eine Tabellendarstellung zunehmend unübersichtlicher, und liegt keine Hilfsdarstellung vor, so wird man bald das Suchen des richtigen Zustandes in den Tabellen aufgeben. Dies betrifft natürlich auch o.g. array-basierte Automatentafeln.

1.2 Projektziel

Um diese Probleme zu lösen (indem man sie elegant umgeht), und um generell "mal einen anderen Turingsimulator" zu entwickeln, der sich von den im Internet Verfügbaren unterscheidet, bietet sich eine Darstellung der Zustände als Transitionsgraph an. Solch ein Simulator hat den Vorteil, dass man die Zustände (bzw. die geometrischen Formen, mit denen sie dargestellt werden) nach eigener Vorstellung in zwei Dimensionen frei anordnen kann, was bei Tabelleneinträgen nicht der Fall ist.

Im Rahmen dieser Arbeit wird daher ein eigener Turingsimulator mit integriertem "Designer" entwickelt. Ganz klar wird dabei die grafische Oberfläche den größeren Anspruch gegenüber der Datenstruktur zum Zustandsgraphen oder zur Turingmaschine einnehmen, was aber unvermeidbar ist.

Der fertige Turingsimulator ist unter die GNU Lesser/Library General Public License [LGPL] gestellt und kann unter `http://jengelh.hopto.org/` [jengelh] heruntergeladen werden. `(Liegt natürlich auch der Arbeit bei.)`

[1] Beispiel siehe [StatArray]

[2] Unix- und Linux-Systeme installieren diesen gleich mit — oder haben ihn zumindest auf der gleichen CD / dem gleichen CD-Set.

[3] Die bekannten `.so`- oder (noch bekannteren) `.dll`-Dateien

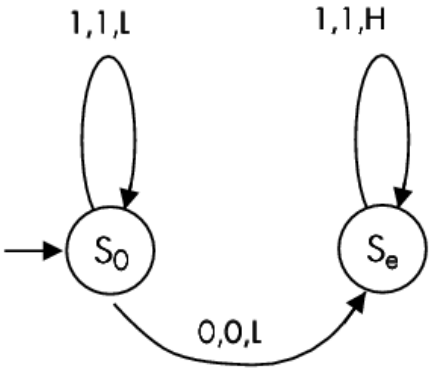

Abbildung 1: Beispiel für einen Transitions-/Zustandsgraphen

2 Grundlagen

2.1 Anforderungen an den Simulator

Das Primärziel dieses Simulators ist, wie schon erwähnt, dass die Definition und Darstellung der Zustände durch Zeichnen erfolgt, statt sie in einer Tabelle einzutragen. Das Sekundärziel wird die Kopplung von Turingautomaten sein.

Zu den Elementarfunktionen, die wichtig für den Betrieb des Simulators sind, gehören in dieser Kategorie die Möglichkeit, die Turingmaschine manuell, aber auch durch einen Taktgeber automatisch zu steuern. Auch soll es die Möglichkeit geben, Graphen abzuspeichern und wieder zu laden, was auch für die Kopplung notwendig ist.

Einige optionale Zusatzfunktionen haben nur eine unwesentlich größere Implementationszeit als wenn man sie auslassen würde. Z.B. führen CPUs Operationen bevorzugt auf Bytes (und Multibytes) statt einzelnen Bits aus[4], und die meisten Programmiersprachen folgen diesem Prinzip[5]. Es empfiehlt sich daher, dass das Turingband auf Bytes statt einzelnen Bits operiert. Das erhöht zugleich die Kapazität des Eingabealphabets.

Zwei andere Punkte, die für mich wichtig sind, ist erstens die Portabilität, und zweitens die freie Verfügbarkeit des Codes auch außerhalb dieser Arbeit.

2.2 Realisierung

Für den Designer wird eine grafische Oberfläche notwendig, wie in Kapitel 3.1 erläutert. Ich verwende dazu [wxWidgets], eine portable Grafikbibliothek, die je nach unterliegendem Grafikset das Aussehen verändern kann. Außer dass mir die APIs von [GTK] und [Qt] nicht geläufig sind, hat WX schon mehr Erfahrung mit der anderen Welt (Win32). WX ist seit seinen Anfängen 1992 dabei [wxIntro], GTK seit 1998 [GtkW32], und Qt wird erst noch für Mitte 2005 erwartet [QtW32]. GTK ist darüber hinaus in C geschrieben, was ich wegen fehlendem objekt-orientiertem Design ungern in der Grafik verwende. Qt (in C++) hat sich (bei meiner früheren Verwendung von [KDE]) bezogen auf Größe, Speicherverbrauch und Geschwindigkeit als Schwergewicht herausgestellt.

[4] Um ein Bit zu ändern, muss mindestens ein ganzes Byte gelesen und dann wieder zurückgeschrieben werden. (`var = var ^ 4`)

[5] C kennt darüber hinaus Bitfields, die sich wie elementare Typen verhalten (`viertes_bit = 1`), nach Kompilation aber auch als CPU-Anweisung "`xor byte ptr [var], 4`" vorliegen.

Der Simulator dagegen hat kaum Bedürfnisse und könnte daher auch CLI[6]-textbasiert geschrieben werden. Ein CUI[7] zu entwerfen, würde erstens den Zeitrahmen sprengen, und zweitens hat ein Turingsimulator für die Hauptbenutzer von CLIs/CUIs, Serveradministratoren, nicht viel Nutzen. Da schon ein grafischer Teil wegen des Designers vorhanden ist, wird auch der Simulator grafisch implementiert. Die Datenstrukturen werden aber so entworfen, dass problemlos ein CLI-Simulator dem Source-Tree hinzugefügt werden könnte, oder andere nicht-grafische Anwendungen diese nutzen können. Als Parser für zu ladende Dateien empfiehlt sich [libxml], den ich schon früher für andere Projekte benutzt habe.

Die Frage nach der Programmiersprache erübrigt sich fast, nachdem die verwendeten Kernkomponenten vorgestellt wurden. Java fällt von vornherein wegen Mangel an Flexibilität bezüglich Datenstrukturbehandlung[8] heraus. wxWidgets bietet zwar nicht weniger als zwei externe Interfaces für Perl und Python an, meine Wahl fällt aber auf C++, da es vollständig kompiliert wird[9] und es Polymorphismus[10], Overloading sowie Templates in C++ gibt. Besonders Vererbung trägt bei einigen der angedachten Datenstrukturen zu einem kürzeren Code bei, als wenn sie in ANSI C geschrieben wären.

2.3 Vergleich mit anderen Turingsimulatoren

Die ersten beiden Kapitel deuteten indirekt schon an, dass es im Netz hauptsächlich tabellenbasierte Simulatoren gibt. Im [DMOZ] gibt es zwar Einträge zu Alan Turing, nicht aber für Turingsimulatoren. Besuchen wir also die Startseite eines namenhaften Suchdienstes, [Google], und fragen nach "Turing Simulator". Das [Google Directory] bietet in etwa die gleichen URLs.

Auf der ersten Ergebnisseite, [Tril], findet sich ein Java-Applet, wobei die Programmierung der Turingmaschine durch ein Textfenster (unten links) erfolgt, was noch fehleranfälliger ist als es Tabellen sind, da man sich nicht nur in den Zeilen, sondern auch in den durch Kommata getrennten Feldern, verfangen kann. Zur Demonstration gibt es auch einige statisch einkompilierte Programmierungen. Einzelschritt und automatischer Durchlauf mit Geschwindigkeitseinstellung (Step / Start / Speed) sind vorhanden. Abspeichern und Laden von eigenen Maschinenprogrammierungen sind aufgrund der Natur von Webapplikationen[11] nicht möglich.

Auf einer anderen Seite, [Bertol], gibt es ein weiteres Java-Applet, dass einen spezifischen Automaten implementiert — der URI und der Verhaltensweise zufolge ein Turing-Biber[12]. Auch dieser Simulator ist in seiner Funktionalität beschränkt — wahrscheinlich, weil es nur eine Demoversion sein soll. Ausschließlich Debugfunktionalität ("single"-Button) ist verfügbar, eine Reprogrammierung dagegen nicht.

Diese beiden Simulatoren sind eigentlich nicht mit meinem Vorhaben vergleichbar, da sie (ihrem Aussehen und Funktionalität zufolge) eines dieser "kurzlebigen" Programme aus Unterricht / Studium sind und nicht für komplexere Anwendung konzipiert sind. Anders sind die folgenden zwei Simulatoren, die während dieser Arbeit auf Seite 1 von Google (sozusagen die Top 10) aufgestiegen sind. Einer davon ist der MPG-Turing-Simulator [MPGTS], ein textgrafischer (CUI) Simulator für DOS16. Die menügesteuerte Bedienung ist sehr intuitiv und lässt sich auch ohne Lesen der README-Datei problemlos herausfinden (vorausgesetzt man weiß, was

[6] Command Line Interface, siehe [FourUI].
[7] Textgraphical Command(-line) User Interface, siehe [FourUI].
[8] Es gibt nur Values und Object-Pointer.
[9] Java und Perl erzeugen zwar sog. Bytecode, aber auch dieser muss erst noch interpretiert werden.
[10] Verwendung einer abgeleiteten Klasse, wo eigentlich eine Basisklasse erwartet wird
[11] Aber nur im Idealfall — [mskb]
[12] Erläuterung zu Bibern in [VlinTuring]

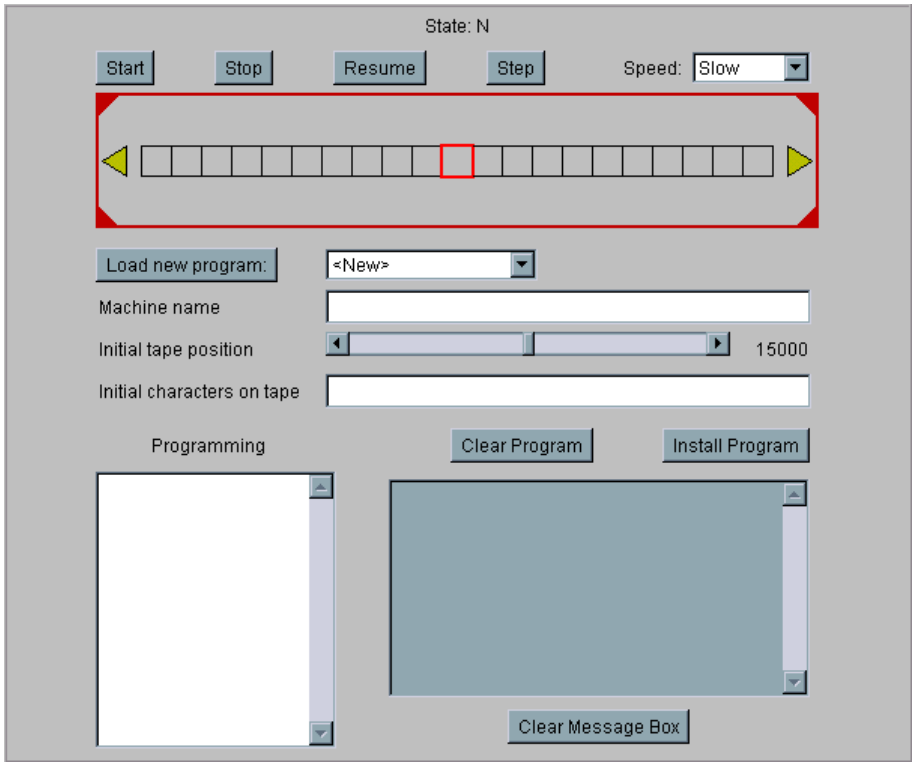

Abbildung 2: Trils Turingsimulator

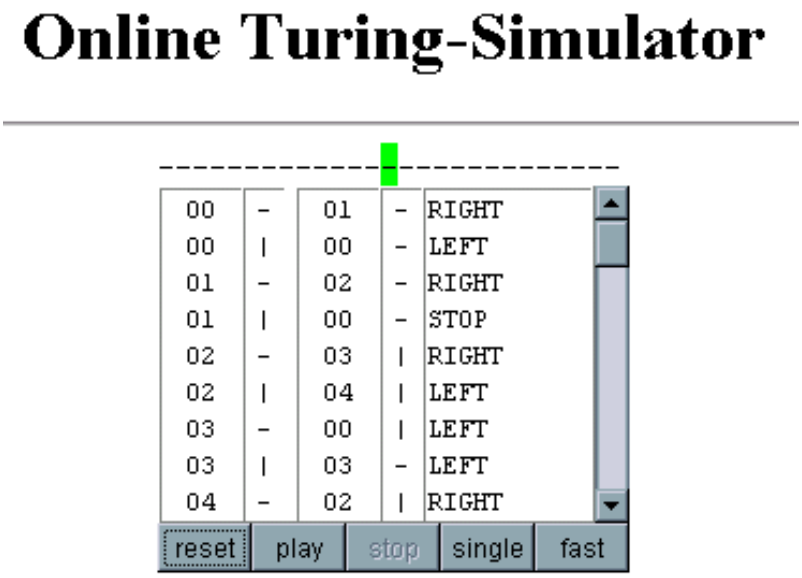

Abbildung 3: “Online Turing-Simulator”

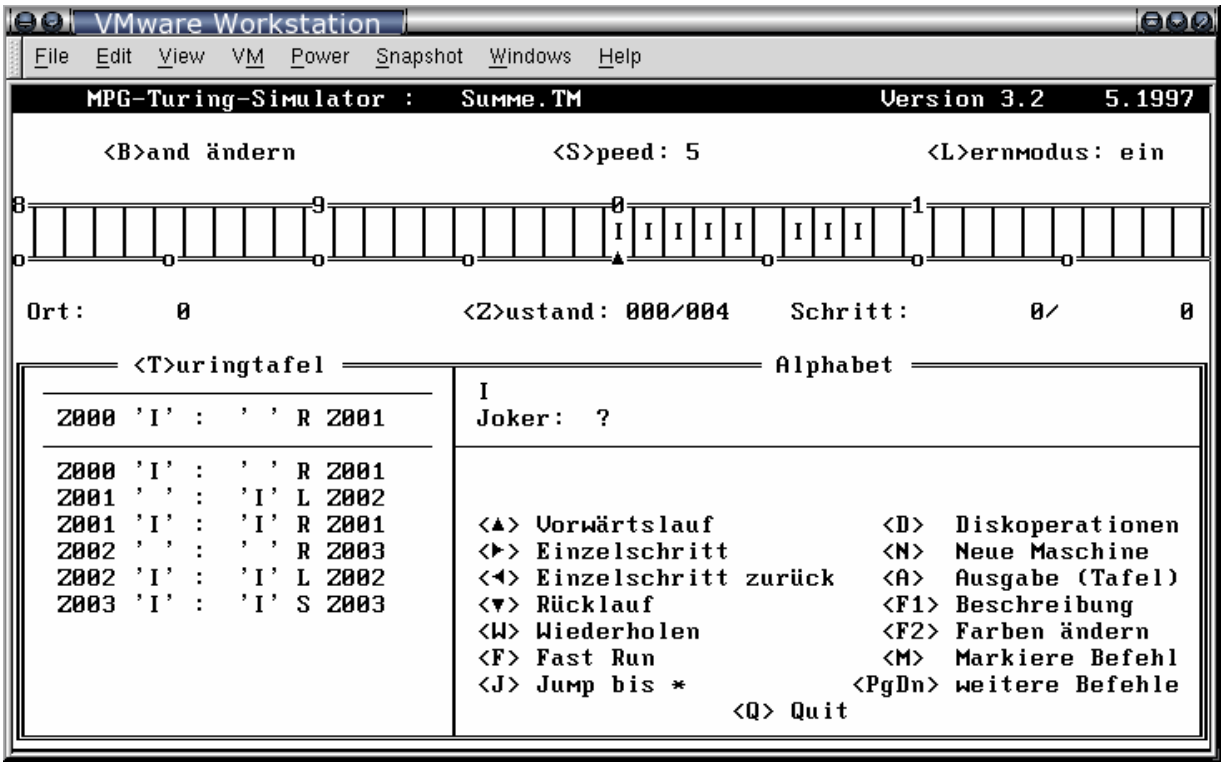

Abbildung 4: MPG-Turing-Simulator

ein Turingsimulator erwartet). Er besitzt keine statischen Programmierungen, hat dafür aber sehr viele (132) Beispiele, die sogar über das in Stufe 12 Behandelte hinausgehen. Es können sogar mehrere Bänder gleichzeitig eingerichtet werden, dennoch bleibt das Kernelement eine Zustandstabelle — was aber aufgrund des Interfacetyps das maximal mögliche ist.

Der letzte Vergleichssimulator, Visual Turing [VisTuring], versucht, ähnlich meiner Vorstellung, auch seinerseits Zustände und Aktionen grafisch darzustellen, tut dies aber in einer unüblichen Weise. Meiner Interpretation des Interfaces nach besteht ein Schritt in Visual Turing aus ⟨Bewegung, Lesen und Schreiben⟩ (in dieser Reihenfolge), was ein substantieller Unterschied zu den in Abbildung 1 dargestellten Graphen ist, welche die Anordnung ⟨Lesen, Schreiben, Bewegen⟩ hätten. Automatische und manuelle Taktgabe sind möglich, Abspeichern und Laden ist kein Problem, zur Debugsteuerung gehört auch Rückwärtsschreiten. Es gibt Verhaltensähnlichkeiten zu S-, I- und LNodes; die Kopplung von Automaten ist möglich, und es kann auch in diese zum Debuggen "eingetaucht" werden. Der Simulator ist allerdings eine Mogelpackung, da es 26 Variablen gibt, in denen Zeichen unabhängig vom Zustand gespeichert und wieder gelesen werden können.

3 Implementation

3.1 User Interface

Als erstes stellt sich immer die Frage, welchen Interface-Typ man verwenden möchte, und ob der damit verbundene Zeitaufwand gerechtfertigt ist, bzw. im Zeitrahmen erfüllt werden kann. Letztendlich ist die Wahl nicht schwer gefallen; ein GUI entsprach der Erwartung. Der Hauptgrund ist, dass die Textkonsole einfach nicht für einen Turing *Designer* — praktisch eine Zeichnungsumgebung — ausgerüstet ist. Mit einer Höhe / Breite von 80x25 Zeichen (maximal

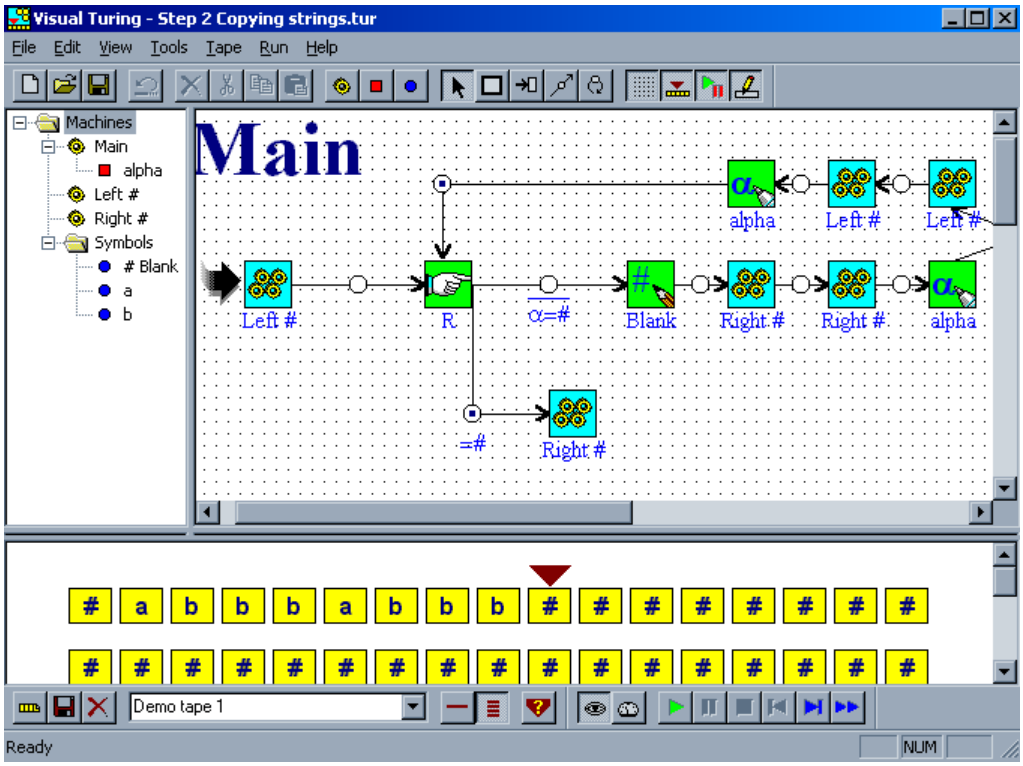

Abbildung 5: Visual Turing

132x60) ist sie auch für einfache Zustandsgraphen zu klein. Ein Kreis mit Sequenznummer wäre schon mindestens 3x3 Zellen groß.

Gleich zu Anfang wurden auch Scrollbalken in das GUI implementiert, um ein bisschen Platz auf der Zeichenfläche zu haben, da schon ein einfacher Graph wie `adder.xml` das Feld füllt, sofern nicht das Anwendungsfenster vergrößert wird. Die virtuelle Größe[13] für den Drawpad[14] sind auf 4096x4096 Pixel festgelegt, können aber ohne weitere Änderungen an einer Stelle Code (`de_pad.cpp`, `DA_WIDTH`) erweitert werden.

3.2 Datenstrukturen zum Graphen

Zu einem typischen Zustandsgraphen gehören die Zustände (als Kreise) und Pfeile für die möglichen Wege. Die Klasse TGraph (`gr_tgrap.cpp`) repräsentiert einen solchen Graphen. Die Zustände sind dabei in einem Array gespeichert. Das ist auch die einzige Möglichkeit, da Überführungsfunktionen entweder nur statisch einkompiliert sein können, oder einen komplexen Parser erfordern würden, wenn sie vom User eingegeben oder aus Dateien geladen würden. Ein (expliziter) Anfangszustand sollte natürlich nicht fehlen, ist aber von der später vorgestellten Klasse TuMach nicht zwingend verlangt.

Da es mehr als nur bloße Zustände geben wird, sei hier nun der Begriff Node eingeführt, der alles abdeckt, was es in einem Graphen gibt. Für die verschiedenen Sorten von Nodes wird eine abstrakte Basisklasse benötigt, damit Polymorphing und Virtual Calls[15] funktionieren.

Der wichtigste Nodetyp ist die SNode (state node), die einen Zustand repräsentiert. Auf dem Drawpad wird sie als roter Kreis dargestellt. Von einer SNode können mehrere Pfeile aus-

[13] Gesamtgröße, einschließlich temporär unsichtbarer Bereiche
[14] Bezeichnung für die weiße Zeichenfläche im Turing Designer
[15] Aufrufen der Funktion einer abgeleiteten Klasse, auch wenn eine Basisklasse verwendet wird

gehen, d.h. sie kann mehrere Folgezustände haben. Die Tabelle der möglichen Folgezustände (Jump-Table) wird mit einem A+B/RB-Tree[16] aus [libHX] realisiert. Die Eingabezustände einer SNode werden dabei jeweils einer Klasse ATarget mit den Feldern ⟨Ausgabezeichen, Bewegung, Folgezustand⟩ zugeordnet.

Für die Kopplung von Turingautomaten ist eine Parallelklasse zu SNode erforderlich, die statt einer Jumptable genau einen Pointer auf einen anderen Graphen besitzt. Die dazu eingeführte INode steht (in diesem Zusammenhang[17]) für "import node" und wird im GUI als blaues Rechteck angezeigt. Da bei Erreichen des Endzustandes im importierten Graphen die Ausführung nicht beendet ist, sondern im übergeordneten Graphen weitergeht, muss es auch für die INode als Ganzes einen Folgezustand geben.

Um in einem Pfeil ein Verbindungsstück einzufügen, sodass er "um die Ecke" gehen kann (auch schon bei einfachen Automaten durchaus notwendig), wird einfach eine Node für den Umweg, statt einer pfeilspezifischen Datenstruktur, verwendet. Eine LNode (link node) dient allein dem Zweck, auf die nächste Node zu verweisen.

3.3 Datenstrukturen zur Turingmaschine

Da Turingmaschinen anforderungslos sind, keine Register o.ä. besitzen und als einzigen Speicher ihren Zustand haben, lassen sie sich daher in wenigen Codezeilen unterbringen. Die Header-Datei `tu_mach.hpp` (faktisch ein Inhaltsverzeichnis der Klassenmethoden) ist sehr kurz. Effektiv sind es gerade mal zwei Funktionen, die zum elementaren Betrieb einer Turingmaschine der Klasse TuMach notwendig sind: `tick`() und `reset`(), jeweils um einen Einzelschritt zu vollziehen und um die Maschine wieder auf den Anfangszustand zu setzen.

Trotzdem ist die `tick`()-Funktion in `tu_mach.cpp` doch etwas länger als zwei Bildschirmseiten (à 25 Z.). Das ist je nach Implementation der Zustandstabellen unterschiedlich — in einem älteren Turingsimulator von mir, der seine Daten noch aus statisch einkompilierten Zustandstabellen bezieht, ist die `tick`()-Funktion gerade mal zwei Zeilen lang. Was `TuMach::tick`() so lang macht, ist generell das Lesen/Schreiben vom/auf Band, welches im alten Turingsimulator in anderen Funktionen eingebaut war. Die rekursive Ausführung von INodes verlangt auch ein paar Zeilen, wie der Tatsache, dass es im Turing Designer nicht nur einen Nodetyp, sondern drei gibt. Codestil und Kommentare kommen ergänzend hinzu.

Die Klasse TuTape (`tu_tape.cpp`) repräsentiert ein Byteband, das nach rechts hin unendlich lang entsprechend der Speicherkapazität des Computers ist. Zwar gehört ein Lese-/Schreibkopf zu einer Turingmaschine, aber nach dem Prinzip des objekt-orientierten Programmierens liest ja nicht der Kopf vom Band, sondern das Band wird angewiesen, sich zu lesen. Diese Kleinigkeit wird beim Programmieren irrelevant: Gleichgültig, ob man nun `tape[5]` (bei `const char *tape`) oder `tape->read(5)` (bei `struct tape *` oder einer Klasse) verwendet, die *Anfrage* muss im Code zur Turingmaschine stehen.

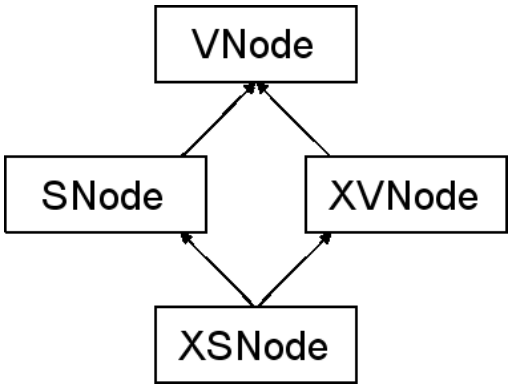

Abbildung 6: The Dreaded Diamond

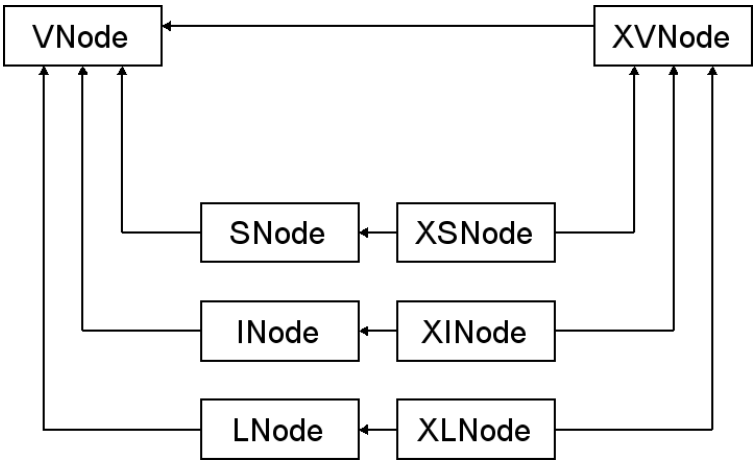

Abbildung 7: Vererbungsschema zwischen den verschiedenen Nodetypen

3.4 Datenstrukturen im GUI

Bei der Einführung eines GUI sind weitere Komponenten in den bereits vorhandenen Strukturen notwendig. Dazu gehören z.B. Koordinaten und optische Darstellung für eine Node.

Zunächst wird eine generische grafische Node XVNode (`de_data.hpp`) benötigt, die von VNode erbt und grafikrelevante Daten hinzufügt. Auch von XVNode werden wieder drei abgeleitete Klassen, für S-, I- und L-Nodes, gebraucht, welche die Eigenschaften von sowohl XVNode als auch bspw. SNode haben. Daraus folgt, dass XSNode von SNode *und* XVNode erben muss[18]. Das Problem des "dreaded diamond" [CFL], wobei mehrmals eine Basisklasse, hier VNode, geerbt wird, lässt sich durch ein zusätzliches Schlüsselwort im Code lösen. (Dies stellt kein Problem dar, wird aber gelegentlich zu Unrecht kritisiert [CFL], 25.3.) Der vollständige Vererbungsgraph sieht dann wie in Abbildung 7 aus.

Auch für die GUI-Komponente des Turingbandes habe ich mich wieder der Mehrfachvererbung bedient. Einerseits soll die Klasse SimTape (`si_tape.cpp`) die gleichen Eigenschaften haben wie ein TuTape, darüber hinaus aber Änderungen am Band gleich im GUI darstellen. Der andere Weg wäre, die Methoden über einen Pointer zum Eltern- oder zu einem Schwesterobjekt zu

[16]Kurz für associative-array-style binary red black tree; Es handelt sich dabei um einen modifizierten Red-Black-Binärbaum, der um die Funktionalität erweitert wurde, nebst einfachen Werten auch Wertepaare zu speichern, ähnlich wie es bei den Hashes (auch Assoziative Arrays genannt) aus Perl der Fall ist.

[17]Die Abkürzung steht in der Unix-Welt eigentlich für "information node" und bezeichnet den Inhalt und Metainformationen wie Zugriffsrechte einer Datei.

[18]Ein Mechanismus ähnlich des "implements xyz" aus Java wäre in C aufwendiger.

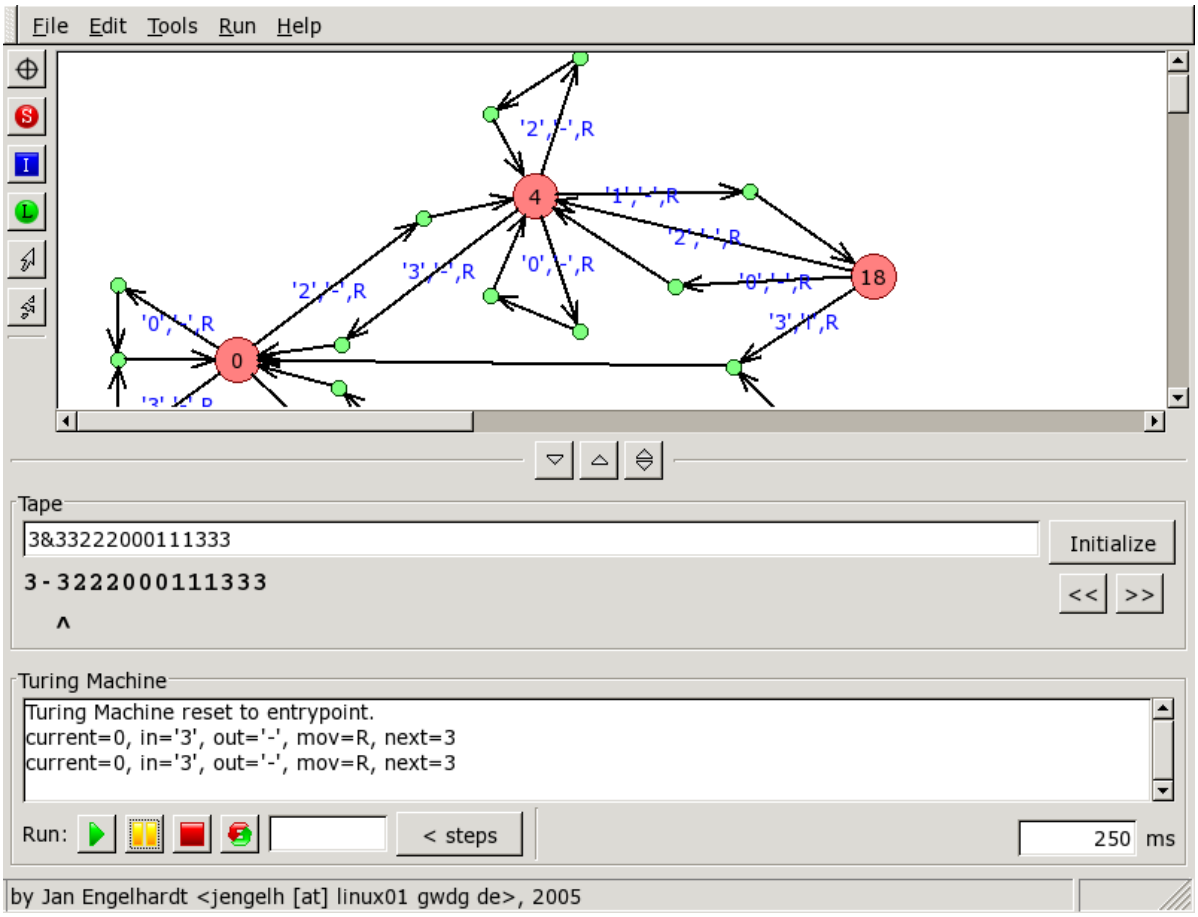

Abbildung 8: Turing Designer, v0.50 / 2005-03-14

delegieren. Das ist aber nicht notwendig. SimTape erbt einfach zusätzlich noch von wxPanel, wodurch es selbst den Status einen "packbaren" Grafikobjekts (ein sog. Widget) erhält.

Bei der Klasse Simulator (`si_sim.cpp`) handelt es sich, ähnlich wie MainWindow (`xf_main.cpp`), nur um einen packbaren Container für die Widgets und Extrafunktionen, die nicht selbst von einer Turingmaschine ausgeführt werden, darunter der automatische Taktgeber und z.B. "nur X Takte" ("<steps"-Button) und "bis Zustand X".

Abschließend noch ein Screenshot vom fertigem Turing Designer:

4 Beispiele

4.1 Addierer

Ein Designer hat hauptsächlich den Vorteil, dass man Objekte so einfügen und anordnen kann, wie man es gewohnt ist, oder zumindest machen würde. Beim Turing Designer ist das nicht anders. Nachdem die Kernelemente — von Nodes bis Targets — vorgestellt wurden, dürfte es nicht schwer sein, den in Abbildung 9 dargestellten Graph im Designer zu reproduzieren. Falls doch, so kann man die Datei `adder.xml` über File▷Open geöffnet werden, um an den Graphen zu gelangen. Es handelt sich dabei um einen Addierer, der auf unär codierten Zahlen arbeitet [VlinTuring], S.3 — auch wenn der Turing Designer volle Bytes (üblicherweise 8 Bit) unterstützt und eine unäre Codierung daher eher spartanisch wirkt.

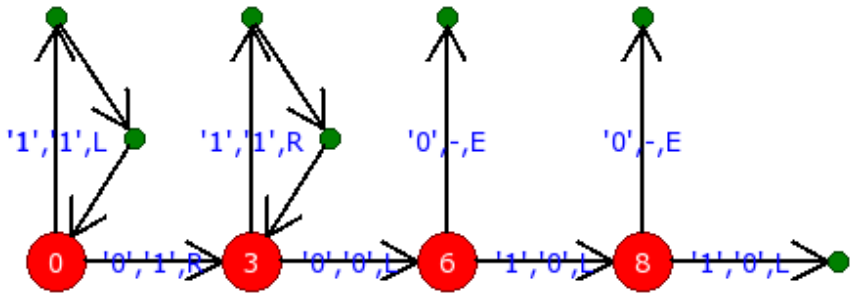

Abbildung 9: Transitionsgraph für den unären Addierer

Im Eingabefeld gibt man z.B. "`0011101111&100`"[19] (2 und 4) ein und klickt auf "Initialize", um ein neues Band "einzuhängen". Durch einen Klick auf den gelben Pause-Button wird ein Takt generiert, mit dem grünen Run-Button wird der Takt von einem Timer gegeben. Der Zeitabstand ist durch das Eingabefeld unten rechts veränderbar. Zu Debugzwecken wird zwischen den Takten der aktuelle Status plus das, was als nächstes geschehen würde, angezeigt. Unser Addierer würde für ein einfaches "`001101&100`"-Band (1+1) demnach folgende Schritte unternehmen:

Turingband/-kopf vor MOV:	Turing Machine reset to entrypoint.
001101100	`current=0, in='1', out='1', mov=L, next=0`
001101100	`current=0, in='1', out='1', mov=L, next=0`
001101100	`current=0, in='0', out='1', mov=R, next=3`
001111100	`current=3, in='1', out='1', mov=R, next=3`
001111100	`current=3, in='1', out='1', mov=R, next=3`
001111100	`current=3, in='0', out='0', mov=L, next=6`
001111100	`current=6, in='1', out='0', mov=L, next=8`
001111000	`current=8, in='1', out='0', mov=L, next=10`
001110000	Reached exit gate. Machine completed.
	Turing Machine reset to entrypoint.

Auf der linken Seite ist das Turingband dargestellt, bevor die `tick`()-Funktion aufgerufen wird. Die Zelle, über der sich der Kopf befindet, ist hier unterstrichen dargestellt. Auf der rechten Seite sieht man die Kontrollausgaben, wie sie bei den verschiedenen Zuständen ausgegeben werden.

Es wird dabei die Bewegung und der Folgezustand angezeigt, die mit dem *nächsten* Aufruf von `tick`() ausgeführt werden würde. Das ist das Essentielle an Debuggern — so können Fehler bereits einen Schritt früher identifiziert werden, Datenstrukturen während dem Debuggen noch geändert werden und man muss nicht von vorne zu debuggen beginnen.

4.2 Gekoppelter Addierer

Um einen gekoppelten Addierer aufzusetzen, braucht es nur zwei INodes, die beide `adder.xml` als Importgraphen beziehen (siehe Abb. 10). Initialisiert man das Band mit drei unär codierten Zahlengruppen, z.B. "`001101101&100`", so kommt nach einem Schritt das Ergebnis "`00111&100`" heraus.

[19] Mit `&` wird die Anfangsposition des Kopfes festgelegt. `&` fällt somit aus dem Eingabealphabet heraus — das ist auch sinnvoll, da `&` in den Grafikroutinen von wxWidgets (u.a.) anderweitig verwendet wird.

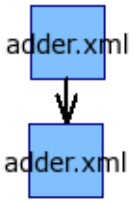

Abbildung 10: Gekoppelter Addierer

Diese Ausführung dieses Graphen ist bereits nach einem Takt zu Ende. Das liegt an der `tick`()-Funktion, die bei Wechsel in den nächsten Zustand sich selbst noch mal aufruft, sofern die nächste Node keine `SNode` ist. Effektiv sind damit Strecken zwischen zwei SNodes in einem Takt ausführbar, was insbesondere bei LNodes, die ja nur als Wegpunkt für den Benutzer dienen, sinnvoll ist. Das gilt auch für INodes, die somit im Zuge eines Taktes transparent ausgeführt werden. Mit einigen magischen Werten für Eingabe- und Ausgabezeichen kann erreicht werden, dass z.B. unmittelbar eine Addition (und ein Zustandswechsel) vollzogen werden, wenn das Eingabezeichen auf einen Target passt. Weiterhin ist das GUI noch nicht in der Lage, in importierte Graphen "einzutauchen", also beim Auftreffen auf die INode statt des aktuellen Graphen die Datei `adder.xml` im Drawpad anzuzeigen. Es würde daher wenig Sinn machen, bei INodes anzuhalten.

5 Rückblick

Auch wenn der Turing Designer ein funktionaler Simulator ist und seine Anforderungen erfüllt, gibt es dennoch einige Punkte, die nicht möglich sind, bzw. nicht in der vorgegebenen Zeitspanne realisiert werden konnten.

Beim Debugging z.B. kann kein Einzelschritt ("Step") durch INodes gemacht werden, wie es bei [VisTuring] der Fall ist. Der importierte Automat wird als atomare Einheit ausgeführt, was eigentlich einem "Next" gleichkommt.

Eine Generalisierung des Grapheninterfaces (`gr_data.hpp`), also dass es für alle Sorten von Automaten verwendet werden kann, nicht nur Turingmaschinen, ist aufgrund der verschiedenen Parameter von Automaten nicht möglich. Eine Turingmaschine müsste ⟨Eingabezeichen, Ausgabezeichen, Bewegung, Folgezustand⟩ speichern, während es z.B. bei einem BCD-Validator ⟨Eingabezeichen, Ausgabestring, Folgezustand⟩ wären. Die gemeinsamen Elemente sind so gering, dass keine sinnvolle Klasse aufgestellt werden kann.

Zuletzt steht im Turing Designer die Tabellenform nicht als Alternative zur Verfügung. Auch wenn ich sie als Hauptproblempunkt bezeichnet habe (und somit ein Einbau eher widersprüchlich wäre), ist das Vorhandensein von mehreren Möglichkeiten dennoch sicher sinnvoll. Spätestens beim Import bzw. Export eines Zustandsgraphen sind Routinen für tabellenartige Darstellung bzw. Textdateien notwendig.

Die Menge von vollwertigen Turingsimulatoren, zu denen ich [MPGTS], [VisTuring] und auch den Turing Designer selbst zähle, scheint sehr gering zu sein. Alternativen sind nur die unzähligen Java-Applets, welche die beschriebenen Problemen haben. Was übrig bleibt, sind DOS- oder Windows-Programme. Ich denke daher, dass die Entwicklung eines weiteren Turingsimulators — diesmal unter Linux — eine doch große Lücke füllt.

Literatur

[Bertol] Java-Applet zur Demonstration eines Busy Beaver
Michael Bertol, Holger Peterson, Horst Prote, August 1996
http://www.fmi.uni-stuttgart.de/ti/personen/Bertol/beaver/bbb.html und Turing.html http://www.fmi.uni-stuttgart.de/ti/personen/Bertol/beaver/Turing.html

[CFL] C++ FAQ Lite
Marshall Cline, 1999–heute
http://parashift.com/c++-faq-lite/multiple-inheritance.html#faq-25.8

[DMOZ] Webkatalog des Open Directory Project
http://dmoz.org/

[FourUI] The four kinds of UIs
Kategorisierung von Programmen mit User-I/O in vier verschiedene Bereiche
Jan Engelhardt, 2005
http://jengelh.hopto.org/coding/uitypes.php

[Google] Google – Suchmaschine mit PageRanking-Verfahren
http://google.com/

[Google Directory] Googles Webkatalog Directory
http://directory.google.com/

[GTK] GIMP Toolkit
Peter Mattis, Spencer Kimball, Josh MacDonald u.v.a.
http://gtk.org/

[GTK@wiki] Wikipedia-Artikel zu GTK
verschiedene Autoren / kein Datum: laufend erneuert
http://en.wikipedia.org/wiki/GTK

[GtkW32] Erste Nachricht in der GTK-Mailingliste über einen Port nach Win32
Tor Lillqvist, August 1998
http://mail.gnome.org/archives/gtk-devel-list/1998-August/msg00089.html

[jengelh] jengelh's site – Homepage von Jan Engelhardt
http://jengelh.medozas.de/

[KDE] K Desktop Environment
Matthias Ettrich u.v.a., 1996–heute
http://kde.org/

[LGPL] GNU Lesser/Library General Public License (version 2.1)
The Free Software Foundation, Februar 1999
http://gnu.org/licenses/lgpl.html

[libHX] General purpose library for daily usage
Jan Engelhardt, 1999–heute
http://jengelh.medozas.de/projects/libHX/

[libxml] The XML C parser and toolkit XML-Parserbibliothek
Daniel Veillard
http://xmlsoft.org/

[MPGTS] MPG-Turing-Simulator
Ulrich Mayr, Max-Plank-Gymnasium Trier/Österreich, 1988, '89, '94, '97, 2002
http://hsg.region-kaiserslautern.de/faecher/inf/material/berechenbar/turing/simulator/mpg/index.php

[mskb] Java Security Issue Allows Access to ActiveX
Bugreport von Microsoft TechNet, 16. Juni 2004
http://support.microsoft.com/default.aspx?scid=kb;en-us;275609&sd=tech

[ncurses] New Curses, Screen handling and optimization package
Bibliothek zum textbasierten Zeichnen
Zeyd M. Ben-Halim, Eric S. Raymond, Thomas E. Dickey (based on pcurses by Pavel Curtis)
http://gnu.org/software/ncurses/

[perldata] Perl Data Types – Manpage
verschiedene Autoren / kein Datum: lfd. aktualisiert
http://perl.com/doc/manual/html/pod/perldata.html

[Qt] Qt Grafikbibliothek
Trolltech AS Norway, lfd. aktualisiert
http://trolltech.com/products/qt/

[QtW32] Qt 4.0 Announcement, 07. Februar 2005
Trolltech AS Norway
http://www.trolltech.com/newsroom/announcements/00000192.html

[StatArray] Beispiel zu statischen Arrays (als Teil dieser Arbeit; enthalten auf CD; nicht in der HTML/PDF-Variante)

[Tril] Trils Turing Simulator
Suzanne Skinner, Datum unbekannt
http://ironphoenix.org/tril/tm/

[VisTuring] Visual Turing IDE
Christian Cheran, Februar 2001
http://cheransoft.com/vturing/

[VlinTuring] Turingmaschinen
Hans-Georg Beckmann, 2003
http://vlin.de/material_2/Turingmaschinen.pdf
S. 3: Unäre Kodierung
S. 36: Fleißige Biber

[wxWidgets] Cross-platform widget library
Julian Smart, 1992–heute
http://wxwidgets.org/

[wxAPI] Allgemeine Präsentation zu wxWidgets
Folie 8: Schichtenmodell von wxWidgets Julian Smart, 2003

[wxIntro] Einführung in wxWidgets, u.a. Geschichte
verschiedene Autoren / kein Datum: lfd. aktualisiert
`http://wxwidgets.org/intro.htm`

Abbildungsverzeichnis