

Hochschule für Technik und Wirtschaft Dresden (FH)
Fachbereich Informatik/Mathematik

Diplomarbeit

im Studiengang Wirtschaftsinformatik

Thema: Reengineering der grafischen Benutzeroberfläche von SNNS

eingereicht von: Konrad Rosenbaum, Matr.Nr. 6966
eingereicht am: 28. August 2000
Betreuer: Prof. Dr. Heino Iwe

Inhaltsverzeichnis

Inhaltsverzeichnis	i
1 Einleitung und Aufgabenstellung	1
Teil I — Einführung in die Problematik	3
2 SNNS	6
2.1 Einordnung von SNNS	6
2.2 Neuronale Netze	6
2.3 Funktionsweise von SNNS	7
3 Ist-Analyse	8
3.1 SNNS-Kernel	8
3.2 SNNS-User-Interfaces	8
4 Zielstellung	11
4.1 Design- und Programmierziel	11
4.2 Diplomziel	11
4.3 Projektziel	12
Teil II — Randbedingungen	13
5 Open Source Software	16
5.1 Was ist Open Source	17
5.2 Lizenzen	17
6 Qt	19
6.1 Was ist Qt	20
6.2 Signale und Slots	20
6.3 MOC	22

7	Internationalisierung	25
7.1	Mögliche Lösungsansätze	25
7.2	GNU <code>gettext</code>	26
7.3	Morgana und Internationalisierung	26
8	GNU Autoconf	27
8.1	Autoconf aus Nutzersicht	27
8.2	Autoconf für kleine Projekte	28
8.3	Autoconf Erweiterungen	29
8.4	Morgana und Autoconf	29
Teil III — Design		31
9	Konzepte	34
9.1	Client/Server	34
9.2	Das Session-Konzept	35
10	Oberfläche	36
10.1	ControlCenter	36
10.2	Display	37
10.3	Pattern Editor	37
10.4	Diagramme	39
10.5	Online-Hilfe	40
11	Inter-Process-Communication	41
11.1	Konzept	42
12	Server	43
12.1	Kommunikation	43
12.2	Zugriff	43
12.3	Ergänzungen	43
Teil IV — Implementierung		45
13	Inter-Process-Communication	48
13.1	Implementierung	48
13.2	Probleme	49

14 SNNS-Server	50
14.1 Konzept	50
14.2 Paket-Format	50
14.3 Umsetzung im Server	51
14.4 Probleme	52
15 Oberfläche	54
15.1 Architektur	54
15.2 Hauptfenster und Sessions	54
15.3 ControlCenter	55
15.4 Display	55
15.5 Online-Hilfe	58
15.6 Probleme	59
15.7 Fehlende Elemente	60
Teil V — Weitere Entwicklungen	61
16 Inter-Process-Communication	64
16.1 Änderungen	64
16.2 Klassische Features	65
16.3 Objektorientierung	66
16.4 Security	67
17 Server	69
17.1 Morgana und SNNS	69
17.2 Ein eigener Simulator	69
18 Oberfläche	71
18.1 Fertigstellung	71
18.2 Weiterentwicklungen	71
19 Projektorganisation	75
19.1 SourceForge	75
Teil VI — Anhänge	79
A CD-Inhalt	81
B Glossar	82
Literaturverzeichnis	85

Index	87
Selbständigkeitserklärung	89

Kapitel 1

Einleitung und Aufgabenstellung

Aufgabe dieser Diplomarbeit war die Entwicklung einer neuen grafischen Oberfläche für den “Stuttgarter Neuronale Netze Simulator” (SNNS), um die Arbeit mit SNNS zu erleichtern.

SNNS ist ein sehr leistungsfähiger Simulator für neuronale Netze. Die existierenden Nutzerschnittstellen sind jedoch relativ schwierig zu bedienen. Ergonomie spielt daher in diesem Projekt eine sehr bedeutende Rolle. In den jeweiligen Abschnitten wird auf diesen Aspekt näher eingegangen.

Um möglichst flexibel auf Neuerungen reagieren zu können, sollte das neue System so gut wie möglich von der konkreten Implementation des Simulatorenkerns abstrahiert werden. Diese Arbeit beschreibt zum größten Teil Designaspekte des Projektes, da die Implementation eines derartig umfangreichen Programmes naturgemäß sehr langwierig und nicht innerhalb von wenigen Monaten schaffbar ist.

Während der Arbeit an diesem Projekt (Morgana¹) ergaben sich weitere Möglichkeiten, die noch nicht verwirklicht werden konnten, und es zeigten sich zahlreiche Schwächen, sowohl im Kernel von SNNS, als auch in den Lösungsansätzen von Morgana. Ich werde in den entsprechenden Kapiteln näher auf die jeweiligen Probleme und mögliche Lösungen eingehen.

Die Arbeit ist grob in fünf Teile gegliedert:

I Einführung in die Problematik — Analysiert die vorhandenen Programme und Konzepte und leitet daraus eine Konkretisierung der Aufgabenstellung ab.

II Randbedingungen — begründet die Grundsatzentscheidungen vor dem Beginn des Projektes und stellt die entsprechenden Werkzeuge vor.

III Design — beschreibt Lösungsansätze und Konzepte von Morgana.

IV Implementierung — ist eine Einführung in die momentane Implementation von Morgana zum Stand des Abgabetermins.

V Weitere Entwicklungen — beschreibt den weiteren Fortgang des Projektes nach Abschluss des momentanen Entwicklungsschrittes, Lösungsansätze für entdeckte Probleme und Ideen zum Ausbau des Projektes.

¹Der Name wurde einem Science-Fiction Roman von R.N.Charrette entlehnt, es handelte sich um eine “echte künstliche Intelligenz”. (siehe [ShRun])

Teil I
Einführung in die Problematik

SNNS ist einer der flexibelsten und erfolgreichsten Simulatoren für neuronale Netze. Jedoch ist er nur relativ schwierig zu bedienen. Dies führte zu der Idee eine neue Oberfläche für SNNS zu schreiben, um die Potentiale des Simulator-Kernels freilegen und nutzbar machen zu können.

Dieser Teil der Arbeit gibt eine Einführung in die Bedingungen am Anfang der Arbeit und leitet daraus die entsprechenden Zielstellungen ab.

Kapitel 2

SNNS

2.1 Einordnung von SNNS

SNNS (der Stuttgarter Neuronale Netze Simulator) ist ein Werkzeug zur Generierung und Erforschung von neuronalen Netzen. Neuronale Netze sind eine relativ junge Disziplin der sog. Künstlichen Intelligenz.

SNNS unterstützt eine sehr große Anzahl von Netzwerktypen und Lernverfahren.

2.2 Neuronale Netze

Dieser Abschnitt bietet einen stark vereinfachten Überblick über die Funktion neuronaler Netze. Eine ausführliche Beschreibung ist zum Beispiel in [Zell] zu finden.

Neuronale Netze lehnen sich an das biologische Vorbild neuraler Verbindungen an. Sie bestehen aus Neuronen (Units) und Verbindungen zwischen diesen Units (Connections, Links), die Daten in exakt eine Richtung zwischen zwei Units transportieren können.

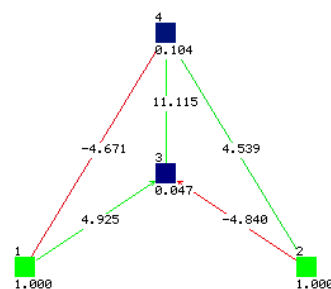


Abbildung 2.1: Ein einfaches FeedForward-Netz

Abbildung 2.1 zeigt ein kleines neuronales Netz. Bei einem solchen Netz werden Werte an die Input-Units (hier: Unit 1 und 2, unten) angelegt, diese modifizieren die Werte entsprechend ihrer Aktivierungsfunktion und leiten sie dann über die Connections an die Hidden- (3) und Output-Units (4) weiter, die Verbindungen multiplizieren den Ausgabewert der Units dabei jeweils mit einem eigenen Wert (Gewicht, eng.: weicht).

Mit entsprechend vielen Units/Connections können dadurch sehr komplexe Funktionen abgebildet (approximiert) werden. Die Aktivierungsfunktionen der Units werden i.d.R. ebenso wie die Struktur des

Netzes vorher festgelegt. Die Werte der Verbindungen werden während der Trainingsphase des Netzes nach einem festgelegten Lernverfahren solange verändert, bis das Verhalten des Netzes hinreichend genau der gesuchten Funktion entspricht oder feststeht, dass die Trainingsphase fehlgeschlagen ist.

In der Auswertungsphase können dem Netz gelernte Eingangswerte vorgelegt werden, die dann den gelernten Ausgangswerten zugeordnet werden, wobei die vom Netz ausgegebenen Werte leicht von den vorher gelernten Werten abweichen können. Werden dem Netz unbekannte Werte vorgelegt nimmt es eine Annäherung der Ausgangswerte vor, die in etwa der gelernten Funktion entsprechen werden.

Ein Vorteil neuronaler Netze ist, dass lediglich typische Eingangs- und Ausgangswerte der Funktion bekannt sein müssen, die Funktion selbst muss nicht bekannt sein.

2.3 Funktionsweise von SNNS

SNNS kann grob in zwei Programmschichten unterteilt werden: User-Interface und Kernel. Momentan existieren zwei unterschiedliche User-Interfaces: batchman — eine Art Shell, in der Simulationen isoliert ablaufen können, und xgui — eine einfache grafische Oberfläche, die allerdings nur schwer zu bedienen ist.

Da Morgana eine eigene Oberfläche implementiert, können wir uns an dieser Stelle auf den Kernel konzentrieren.

Der SNNS-Kernel speichert neuronale Netze in einer relativ flachen Struktur, ähnlich wie relationale Datenbanken, ab. Außer einigen internen Tabellen für die Fähigkeiten des Kernels (Lernverfahren, Aktivierungsfunktionen etc.) existieren drei wichtige Tabellen:

Unit Array speichert alle Units mit allen möglichen Parametern

Sites speichert die Sites¹ aller Units

Links speichert alle Connections (im SNNS: Links), also Verbindungen zwischen den Units.

Der Zugriff auf diese Tabellen und die Simulation erfolgen über Funktionsaufrufe². Diese Funktionen kümmern sich im Hintergrund um den Aufbau und die Administration der internen Tabellen.

Weitere Details zur Nutzung und zur Funktion von SNNS finden sich in [SNNS] und [SNNS-I].

¹Sites kann man sich als eine Art von Sub-Unit vorstellen, die bereits Transformationen auf jeweils einigen der Input-Connections einer Unit vornehmen, bevor die Werte an die Unit weitergeleitet werden.

²Es sind immerhin fast 50 unterschiedliche Funktionen, die für ART-Netze nicht mitgerechnet

Kapitel 3

Ist-Analyse

3.1 SNNS-Kernel

Der SNNS-Kern kann als eigenständige Bibliothek in eigene Programme eingebunden werden. Zugriff auf diese Funktionalität erhält man durch Funktionsaufrufe. (Siehe auch Kapitel 2.1)

SNNS bietet ein sehr flexibles Netzwerk-Modell und eine Vielzahl von Algorithmen, die auf das jeweilige Netzwerk angewendet werden können. Es liegt also durchaus nahe, den Kernel unverändert beizubehalten und die vorhandene Funktionalität zu nutzen.

3.2 SNNS-User-Interfaces

SNNS besitzt zwei unterschiedliche Nutzerschnittstellen. Batchman, eine Shell mit einer Pascal-ähnlichen Syntax, und xgui, eine grafische Oberfläche.

Die SNNS-eigene grafische Oberfläche setzt direkt auf X-Window auf (siehe Abbildung 3.1). Sie hat jedoch einige sehr unangenehme Eigenschaften, die die Arbeit mit dieser Oberfläche sehr schwierig gestalten. So wird zum Beispiel das gesamte Programm beendet, wenn ein einzelnes Fenster über den "Schliessen"-Button der Kontrollleiste anstatt über den Button "Done" geschlossen wird. Einige Dialoge zeigen immer nur einzelne Werte anstatt der gespeicherten Wertlisten an, um an weitere Werte heranzukommen muss man sich mühsam durch die Liste klicken.

Es ist also sehr mühsam auf die Funktionalität von SNNS zuzugreifen, da man entweder eine neue Programmiersprache lernen muss (batchman) oder mit einer sehr schwierig zu bedienenden und kaum fehlertoleranten grafischen Oberfläche (xgui) umgehen muss.

3.2.1 Die SNNS-xgui

Die xgui von SNNS baut direkt auf die von X-Window bereitgestellten Bibliotheken auf¹.

Problematisch an der xgui sind die teilweise unerwarteten Verhaltensweisen, die sich zum großen Teil aus dem Aufbau der relativ einfachen zugrundeliegenden Bibliotheken ergeben:

- die xgui beendet sich sofort, wenn man versucht eines der Fenster über die Buttons der Titelleiste zu schließen

¹und zwar: libXt, libXmu und libXaw

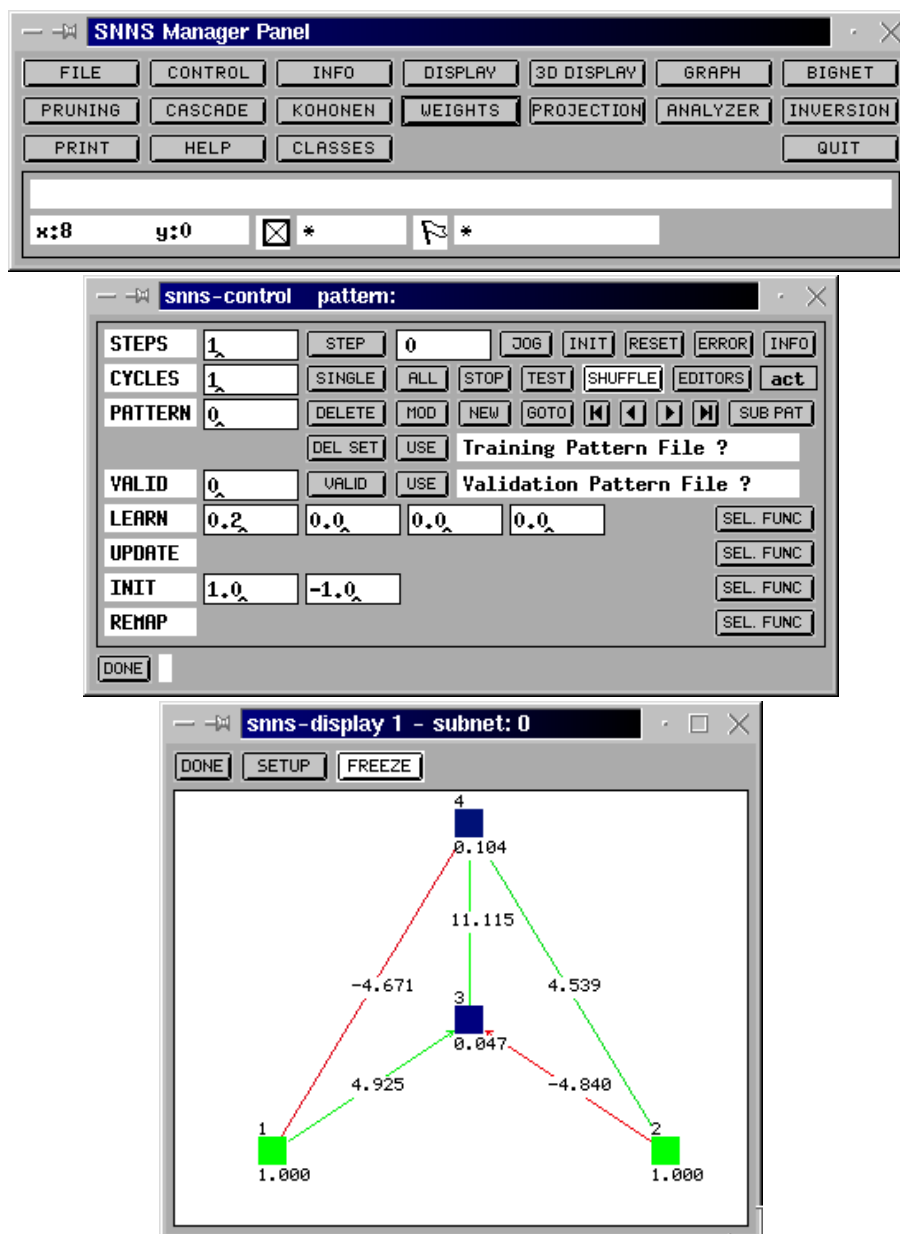


Abbildung 3.1: Die xgui von SNNS

- einfache Buttons werden für sehr verschiedene Aufgaben verwendet, für die normalerweise sehr verschiedene Widgets verwendet werden (in Klammern)
 - Starten und Beenden von Aktionen, Schliessen von Fenstern (Button)
 - Startpunkt für Menüs (Menübalken, Drop-Down-Felder, Listen)
 - Umschalten zwischen Optionen (Checkboxes, Radiobuttons)
- um Kontextmenüs zu öffnen, muss man statt rechter Maustaste `Strg` und die linke Maustaste benutzen, einzelne Ebenen der Kontextmenüs sind nur durch mehrmaliges Benutzen dieser Kombination zu erreichen.

Kapitel 4

Zielstellung

Das allgemeine Ziel der Diplomarbeit und darüber hinaus des Morgana Projektes (siehe Teil 15.7) ist die Behebung der ergonomischen und programmtechnischen Mängel der SNNS-GUI.

4.1 Design- und Programmierziel

Es sollte eine GUI entworfen werden, die sowohl einfach zu bedienen, als auch stabil und gut erweiterbar ist. Das lässt sich so zusammenfassen:

Ergonomie — das Ergebnis sollte möglichst einfach (intuitiv) zu bedienen sein. Die GUI sollte sich dabei auch neuen oder geänderten Bedürfnissen anpassen lassen.

Stabilität — das gesamte System sollte nach Möglichkeit unterbrechungsfrei simulieren können, da jede Unterbrechung einer Simulation potenziell einen erneuten Start erfordern kann.

Fehlertoleranz — das System sollte möglichst tolerant gegenüber Fehleingaben sein und den Nutzer gegebenenfalls darauf hinweisen.

Logische Konsistenz — alle angezeigten Daten sollten auch im Simulationskern gespeichert sein und alle sich im Kern verändernden Daten sollten sich auch in der Anzeige widerspiegeln¹.

Erweiterbarkeit — das System sollte möglichst einfach und flexibel um weitere Netzwerk-Typen und Funktionen erweitert werden können.

4.2 Diplomziel

Ziel der Diplomarbeit ist das Design eines Systems, das die obigen Bedingungen erfüllt. Sowie die Implementation der wichtigsten Bausteine dieses Systems. Da es sich um ein sehr weitreichendes und aufwändiges Vorhaben handelt, kann es nicht innerhalb eines einzigen Semesters verwirklicht werden. Vielmehr sollte während dieser Arbeit eine Grundlage für spätere Weiterentwicklungen geschaffen werden.

¹SNNS speichert zur Zeit nicht, ob ein BigNet-Typ benutzt wurde als das Netz erstellt wurde. Diese Information geht nach einem erneuten Laden des Netzes verloren.

4.3 Projektziel

Am Ende dieser Diplomarbeit wird das Projekt in Open Source übergehen und frei weiterentwickelt werden. Ziel ist dann die Fertigstellung des Systems und Implementation neuer Funktionalität. (Siehe dazu Teil 15.7 dieser Arbeit, ab Seite 63.)

Teil II

Randbedingungen

In diesem Teil meiner Arbeit werde ich grundlegende Einblicke in die wichtigsten Themen geben, die zum Verständnis der Interna des Projektes notwendig sind. Dieser Teil behandelt Grundsatzentscheidungen, die dem Projekt einen Rahmen gesetzt haben.

Da dieser Arbeit sowohl zeitliche Grenzen als auch eine begrenzte Seitenzahl gesetzt sind, ist es unmöglich alle diese Themen umfassend abzuhandeln. Ich werde daher an den jeweiligen Stellen auf weiterführende Literatur verweisen.

Kapitel 5

Open Source Software

Projekte von der Größenordnung von Morgana (mehrere Mann-Jahre Arbeit) können nicht mehr von einzelnen Personen allein umgesetzt werden, ohne den Anschluss an die aktuelle Entwicklung zu verlieren. Es bieten sich mehrere Lösungen an, die mehrere Personen an das Projekt binden:

Aufspaltung in weitere Diplom-Projekte und Workshops ist die offensichtlichste Lösung, die jedoch den Nachteil hat, dass immer nur eine oder wenige Personen an das Projekt gebunden werden können, die zusätzlich noch unter dem Druck stehen innerhalb einer sehr kurzen Zeit sichtbare Ergebnisse zu liefern. Dieser Ansatz würde aller Wahrscheinlichkeit nach zu einzelnen völlig unabhängigen Entwicklungsschüben führen, die immer nur kleine Teile des Quelltextes betreffen, die Gesamtstruktur jedoch außen vor lassen. Der Vorteil dieser Methode ist die direkte Bindung von akademischem (also Forschungs-) Wissen an das Projekt.

Bindung an eine kommerzielle Entwicklung dürfte kaum realistische Aussichten auf eine gute und schnelle Weiterentwicklung bieten, da es dann einerseits zu Konflikten mit der Lizenzierung von SNNS kommen würde und andererseits der Markt für auf Forschung ausgelegte neuronale Netze Simulatoren bereits durch das originale SNNS besetzt ist. Außerdem hätte diese Lösung den Nachteil an rein monetäre Interessen gebunden zu sein. Experimentelle, risikobehaftete Erweiterungen werden in einer solchen Umgebung nur schwer entstehen.

Eine Weiterführung als Forschungsprojekt hätte wahrscheinlich ähnliche, wenn auch nicht ganz so offensichtliche Auswirkungen. Forschungsprojekte sind immer auf Ergebnisse im Kerngebiet des jeweiligen Projektes ausgerichtet, die Oberfläche würde hier wahrscheinlich zu kurz kommen, was jedoch dem Grund des Projektes zuwiderlaufen würde. Spätestens mit einer Drittmittelförderung des Projektes würden Teile der Funktionalität wahrscheinlich nur noch den Sponsoren zur Verfügung stehen, was eventuell positive äußere Einflüsse stark einschränken würde.

Open Source gibt jedem mit entsprechenden Interessen die Möglichkeit sich an diesem Projekt zu beteiligen. Ein Nachteil ist das Risiko, dass die Entwicklung des Projektes vom guten Willen freiwilliger Entwickler abhängig ist, d.h. wenn das Interesse an diesem Projekt zu gering ist, wird es sich eventuell nur sehr langsam oder gar nicht weiterentwickeln. Der zweite (scheinbare) Nachteil ist, dass damit die Kontrolle über große Teile der Entwicklung an unbekannte Personen abgegeben wird. Man kann jedoch annehmen, dass genügend Experten existieren sollten, die sowohl das notwendige Wissen für eine Projektbeteiligung als auch den Willen dazu haben; sollte dies nicht so sein, dann ist anzunehmen,

dass auch das Interesse an einer Nutzung des Systems nur gering ist¹. Der zweite Nachteil wird in der Praxis dadurch ausgeglichen, dass es Aufgabe des Projektleiters ist die Entwickler zu notwendigen Entwicklungen zu motivieren. Schließlich hat Open Source den Vorteil von Entwicklern aus nahezu allen Fachgebieten und Nationen profitieren zu können. Trotzdem sollten man die erste Möglichkeit (Diplomarbeiten und/oder Workshops) als Ergänzung in Betracht ziehen, um zusätzlich einiges akademisches Wissen an das Projekt zu binden.

Morgana als Open Source freizugeben, gibt diesem Projekt die Chance auch von den Erfahrungen und dem Engagement von Entwicklern außerhalb des universitären Bereichs zu profitieren. Dadurch kann nicht nur Kompetenz im Bereich der neuronalen Netze (und weiterer KI-Bereiche), sondern auch ein erhebliches software-technisches Know-how in das Projekt einfließen. Inwieweit Morgana tatsächlich von diesen Effekten profitieren kann, bleibt abzuwarten.

Ich werde im Folgenden einen kurzen Überblick über das Wesen von OpenSource-Projekten geben. Dieses Kapitel basiert zum größten Teil auf den Arbeiten von Eric S. Raymond ([ESR99, WWWOS, WWWESR]).

5.1 Was ist Open Source

Open Source oder auch freie Software² ist Software deren Quellen für jeden frei zugänglich sind. Jeder hat das Recht diese Software zu nutzen, frei weiterzugeben und zu modifizieren. Im Gegensatz zu kommerzieller (proprietärer) Software, die weder weitergegeben noch modifiziert werden darf.

Die Idee hinter Open Source (siehe [ESR99]) ist, dass man im Austausch gegen die Quellen der eigenen Entwicklungen das Expertenwissen vieler anderer Entwickler erhält, die die Software nicht nur nutzen, sondern auch testen und oft gleich komplette Patches für die entdeckten Probleme mit oder kurz nach der Fehlermeldung liefern. Auf diese Weise können innerhalb kurzer Zeit hunderte Programmierer auf ein Projekt konzentriert werden. Dadurch können sich Open Source Projekte sehr schnell entwickeln und deren Komplexität kann leicht auf viele Schultern verteilt werden³.

Die Protagonisten dieser Bewegung⁴ haben inzwischen eine (Sub-)Kultur entwickelt, die seit den 90'ern sehr schnell wächst und in der modernen Zivilisation einzigartig ist. Eine genauere Betrachtung dieser Kultur und ihrer Motivationen würde allerdings den Rahmen dieses Kapitels sprengen — ich verweise daher auf die Publikationen von Eric S. Raymond.

5.2 Lizenzen

Da es bei Open Source um Software geht, sind die jeweiligen Lizenzen ein wichtiges Mittel der Kommunikation, da diese ausdrücken, welche Rechte man anderen zugesteht und welche man für sich selbst beansprucht.

Inzwischen gibt es relativ viele Open Source-Lizenzen (siehe [WWWOS]). Bedingungen für eine Open Source-Lizenz sind: sie muss das Recht geben die Software ohne Einschränkung zu nutzen, den

¹Bei Open Source geht man normalerweise davon aus, dass der Kreis der Nutzer und der der Entwickler sich zu einem sehr großen Teil gleichen

²es gab einige Zeit Streit, welcher der beiden Begriffe der bessere sei; "freie Software" ist der ältere und "Open Source" der inzwischen am weitesten verbreitete und von vielen Entwicklern bevorzugte Begriff, ich werde im Folgenden den Begriff "Open Source" benutzen ohne eine Wertung dieser beiden Begriffe vorzunehmen

³Gute Beispiele für beide Effekte sind der Linux Kernel und das KDE Projekt, beide vereinigen auf sich hunderte Programmierer und bewältigen eine geradezu erstaunliche Komplexität.

⁴Ich werde in dieser Arbeit den Begriff "Open Source-Entwickler" nutzen.

Quelltext zu verändern und den geänderten Quelltext weiterzugeben und sie darf keine Einschränkungen machen, wer diese Rechte hat.

Die "Open Source Definition"⁵ beinhaltet die exakten Bedingungen, unter denen eine Lizenz als Open Source-Lizenz bezeichnet werden darf. Die Bedingungen für den Begriff "freie Software" sind teilweise noch restriktiver gehalten⁶, was hier jedoch nicht von Bedeutung ist.

Der Bereich der Open Source Lizenzen reicht von denen, die jedem nahezu jedes Recht einräumen, bis zu denen, die dem Originalautor das Recht einräumen freie Teile in eine "professional version" zu übernehmen und kommerziell zu vermarkten.

Die "BSD-style" Lizenzen sind völlig offene Lizenzen. Sie räumen jedem das Recht ein modifizierte Versionen unter eine eigene Lizenz zu stellen, solange die Namen der Originalautoren noch genannt werden. Diese Form der Lizenzen scheint dazu zu führen, dass Projekte sehr stark zersplittert werden. Bei BSD-Unix hat es dazu geführt, dass nicht nur mehrere freie, sondern auch mehrere kommerzielle Versionen vertrieben werden.

Die General Public License (GPL) und die Lesser General Public License (LGPL) sind die restriktivsten aber auch beliebtesten Vertreter der Open Source Lizenzen. Die GPL stellt jedem das Recht auf Nutzung, Modifikation und Verbreitung des Programms zur Verfügung. Man darf aber keine Copyright-Hinweise entfernen und das Programm muss unter der GPL bleiben. Im Unterschied zur LGPL verlangt die GPL auch, dass jedes Programm, das Teile eines Programms oder einer Bibliothek beinhaltet oder modifiziert ebenfalls unter GPL stehen muss. Als Modifikation wird auch das Linken gegen die Binärform eines Programms oder einer Bibliothek gesehen.

Die Klasse der NPL-artigen Lizenzen⁷ kommt den kommerziellen Lizenzen am nächsten. In der Regel sind sie von der GPL abgeleitet, enthalten aber die zusätzliche Klausel, dass der Originalautor als einziger Modifikationen des freien Quelltextes kommerziell vermarkten darf. Sie werden häufig für ehemals vollständig kommerzielle Produkte beim Übergang in Open Source benutzt.

Die Lizenz von SNNS ist konform zu den Bedingungen der "Open Source Definition". Sie gehört in die Klasse der NPL-artigen Lizenzen und macht die zusätzliche Restriktion, dass Modifikationen nur als Patch weitergegeben werden dürfen⁸.

Die SNNS-Lizenz und die GPL schließen sich gegenseitig aus. Dies ist problematisch, da Morgana gegen einige Quelltexte gelinkt wird, die unter GPL stehen⁹. Um beiden Lizenzen zu entsprechen müssen einige Randbedingungen eingehalten werden. Zum Beispiel dürfen die vorgenommenen Modifikationen an SNNS¹⁰ nur als Patch weitergegeben werden.

⁵der genaue Wortlaut ist in einer aktuellen Version auf [WWWOS] zu finden

⁶nachzulesen auf [WWWFSF]

⁷NPL steht für "Netscape Public License", sie wurde von Netscape bei der Freigabe des Quelltextes des Netscape Navigators eingeführt; die von Qt benutzte QPL stammt ebenfalls aus dieser Klasse, sie wurde nach monatelangen Verhandlungen zwischen Troll-Tech und dem KDE-Team für Qt ab Version 2.0 eingeführt

⁸was externe Bugfixes nach meinen eigenen Erfahrungen nicht gerade einfacher gestaltet

⁹z.B. GNUgettext

¹⁰diese waren notwendig, um SNNS mit der GNU libc Version 2.1 und C++-Code linken zu können

Kapitel 6

Qt

Für den Entwurf einer graphischen Oberfläche ist die Entscheidung für ein GUI-Toolkit essentiell, da dadurch nicht nur das Aussehen sondern auch die interne Strukturierung und oft auch die Programmiersprache festgelegt wird. Im Vorfeld dieser Arbeit wurden einige Anforderungen an das Projekt und damit auch das zugrundeliegende GUI-Toolkit deutlich:

Portabilität — Morgana sollte mit möglichst wenig Aufwand auf möglichst viele verschiedene Plattformen transportiert werden können, die Mindestanforderungen sind dabei Linux, möglichst viele Unix-Derivate und Windows (NT). Damit sind schon alle Toolkits ausgeschlossen, die nur auf einer Plattform existieren (z.B. die MicroSoft Foundation Classes (MFC) und Motif). Die Programmiersprachen sind dadurch ebenfalls auf einige wenige eingeschränkt, die auf allen diesen Plattformen vorhanden sind (z.B. Perl, Tcl/Tk, C, C++).

Einfachheit, Erweiterbarkeit — Programmiersprache und Toolkit sollten in der Lage sein eine übersichtliche Implementation zu fördern, um das Projekt so lange wie möglich und so flexibel wie möglich erweitern zu können. Damit werden die Programmiersprachen auf die höheren und objektorientierten Dialekte beschränkt, wobei sich die objektorientierten Sprachen hier als besonders übersichtlich darstellen.

Geschwindigkeit — das entstehende System sollte sich aus Nutzersicht als “flüssig” zeigen. Damit scheiden Skriptsprachen (Perl, Tcl/Tk) und Bytecodeinterpreter (Java) aus, da sie eine zusätzliche Interpreterschicht benötigen.

Kompatibilität zu SNNS — SNNS muss sich aus der jeweiligen Programmiersprache heraus aufrufen lassen. Diese Bedingung erfüllen inzwischen nahezu alle Programmiersprachen, da sie in der Lage sind externe Bibliotheken aufzurufen.

Konformität zu Open Source — Programmiersprache und Toolkit sollten für Open Source Entwickler verfügbar und erschwinglich sein. Es kommen also nur noch Programmiersprache in Frage, für die es einen freien Compiler/Interpreter gibt (z.B.: C, C++: GNU C; Fortran: GNU g77; Perl). Als Toolkits kommen daher ebenfalls nur Open Source Toolkits in Frage.

Da die meisten Open Source Entwickler C und C++ beherrschen, sind dies die Favouriten unter den Programmiersprachen. Pascal und Fortran werden nur äußerst selten eingesetzt und Perl und Tcl/Tk fallen durch das Geschwindigkeitskriterium aus der Wahl. Andere Programmiersprachen scheinen unter Open Source Entwicklern kaum verbreitet zu sein¹.

¹Konjunktiv, da sich über Open Source kaum objektive Angaben machen lassen: es ist unmöglich die Nutzer von Open Source zu zählen.

Dies sind einige der verfügbaren Toolkits (ich beschränke mich auf die häufig eingesetzten und möglichst vielen Entwicklern bekannten Toolkits):

- Tk** — wurde ursprünglich für Tcl entwickelt (daher heute auch “Tcl/Tk”) und später zu Perl und einigen anderen Sprachen portiert, interessant ist hier das C-Interface von dem aus Tk auch in compilierten Programmen genutzt werden kann. Die Widget-Stacks sind relativ einfach gegliedert und werden durch hierarchische Namen wiedergespiegelt. Problematisch ist hier die relative Empfindlichkeit gegenüber Layoutänderungen, da sich dabei auch die Namen der Widgets ändern.
- GTK** — ein in C geschriebenes Toolkit, das auf X-Window aufsetzt, inzwischen aber auch nach Windows portiert wurde. In dieser Form gilt GTK jedoch als nur schwer handhabbar. GTK wird zum Beispiel in GIMP und Gnome eingesetzt.
- GTK—** — ein C++-Wrapper um GTK, der es erlaubt relativ flexibel und komfortabel mit GTK zu arbeiten.
- Qt** — ein ursprünglich von der norwegischen Firma TrollTech entwickeltes Toolkit auf C++, das sehr flexible Hierarchien von Widgets und anderen wichtigen Objekten aufbauen kann. Qt für X-Window ist noch immer in den Händen von TrollTech, unterliegt inzwischen aber einer Open Source-Lizenz. Qt wird zum Beispiel von KDE eingesetzt. Qt für Windows unterliegt einer kommerziellen Lizenz, was inzwischen aber kein Problem mehr darstellen dürfte, da Teile von XFree86 nach Windows portiert wurden und das freie Qt damit über einen X-Emulator auch auf Windows betrieben werden kann.

Die komfortabelsten Toolkits sind damit GTK— und Qt. Die Entscheidung zwischen diesen beiden Toolkits ist demnach allein von den Vorlieben des jeweiligen Programmierers abhängig. Für Morgana habe ich mich für Qt entschieden, da es ohne zusätzliche Wrapper auskommt und bei Beginn der Arbeit bereits von mir erlernt war. Außerdem besteht damit die Möglichkeit über zusätzliche Klassen die Funktionalität von KDE zu nutzen, was inzwischen der verbreiteste Desktop unter Linux zu sein scheint und sich auch unter anderen Unix-Derivaten einiger Beliebtheit erfreut².

Ich gebe hier einen kurzen Überblick über die Fähigkeiten von Qt. Eine vertiefende Einführung in das Thema findet sich in [QT].

6.1 Was ist Qt

Qt³ ist eine Klassenbibliothek zur Entwicklung grafischer Oberflächen. Es ist sowohl unter Windows, als auch nahezu allen Unix-Dialekten verfügbar. Darüber hinaus hat es einige Fähigkeiten, die die Entwicklung plattformunabhängiger Software weiter erleichtern.

Qt gehört in die Klasse der GUI-Emulationen. Das heißt es bildet seine eigene Schnittstelle auf sogenannte Grafikprimitive der jeweiligen Plattform ab.

6.2 Signale und Slots

Im Gegensatz zu den meisten anderen GUI-Toolkits erweitert Qt den Sprachumfang von C++ um die Elemente “Signal” und “Slot”. Statt umständlich Ereignis-Strukturen durch eine Hierarchie von Objekten zu schicken, hat man damit die Möglichkeit den Sender und den Empfänger eines Ereignisses direkt miteinander zu verbinden.

²Diese Aussage beruht auf Beobachtungen in einer der KDE-Maillisten, in der öfter auch Fragen zu Solaris und anderen Unix'en auftauchten.

³gesprochen wie das englische Wort “cute” (niedlich, schlau)

Dieses einfache Programm nutzt diese Fähigkeit aus, um einen Klick auf den “Beenden”-Button mit dem Schliessen des Fensters zu verbinden:

qtsimple.cpp

```
1 #include <qapplication.h>
2 #include <qpushbutton.h>
3 #include <qwidget.h>
4 #include <qlabel.h>
5
6 int main(int argc, char**argv)
7 {
8     QApplication app(argc, argv);
9
10    QWidget *w=new QWidget;
11    w->setCaption("Simple App");
12
13    w->resize(150,100);
14
15    new QLabel("Simple Label",w);
16
17    QPushButton *p=new QPushButton("Beenden",w);
18    p->move(10,50);
19
20    QObject::connect(p,SIGNAL(clicked()),w,SLOT(close()));
21
22    w->show();
23
24    app.setMainWidget(w);
25
26    return app.exec();
27 }
```

Die Methode `connect` verbindet dabei das Signal `clicked()` des Buttons mit dem Slot `close()` des Fensters (`w`). Sobald mit der Maus auf den Button geklickt wird, erzeugt (im Qt-Sprachgebrauch “emittiert”) dieser das Signal `clicked()`, welches an den Slot `close()` des Fensters weitergereicht wird. Dieser wiederum schließt das Fenster und beendet damit das ganze Programm⁴.



Abbildung 6.1: Einfaches Qt Fenster

⁴Qt Programme beenden sich immer dann, wenn das Fenster geschlossen wird, das mit `setMainWidget(...)` an das Applikationsobjekt übergeben wurde.

6.3 MOC

Will man eigene Klassen mit Signalen und Slots definieren, muss der "Meta Object Compiler", kurz MOC, genutzt werden. Er erzeugt den C++-Code, der nötig ist, um C++ um die Funktionalität von Signalen und Slots zu erweitern⁵. Diese Klassen müssen in jedem Fall von der Klasse `QObject` abgeleitet sein.

Das Beispiel von oben lässt sich nun relativ einfach um eine Eingabezeile erweitern, die den statischen Text von oben ("Simple Label") verändern kann:

qtmoc.h:

```
1 #include <qwidget.h>
2 #include <qstring.h>
3 #include <qlineedit.h>
4
5 class MyClass:public QWidget{
6     Q_OBJECT
7
8     public:
9         MyClass();
10        ~MyClass();
11
12        public slots:
13            void refreshLine();
14
15        signals:
16            void setTextLine(const QString&);
17
18        private:
19            QLineEdit *edt;
20 };
```

qtmoc.cpp:

```
1 #include <qapplication.h>
2 #include <qpushbutton.h>
3 #include <qlabel.h>
4
5 #include "qtmoc.h"
6
7 MyClass::MyClass()
8 {
9     setCaption("Simple App");
10
11    resize(150,150);
12
13    QLabel *lab=new QLabel("Simple Label",this);
14    connect(this,SIGNAL(setTextLine(const QString&)),
15            lab,SLOT(setText(const QString&)) );
```

⁵Speziell sind das die Implementationen einer eigenen RunTime Type Information, eine Liste aller Signale und Slots mit Namen und Zeigern sowie die Implementation der Signale.

```

16
17     edt=new QLineEdit("something",this);
18     edt->move(0,40);
19
20     QPushButton *p=new QPushButton("Setzen",this);
21     p->move(10,70);
22     connect(p,SIGNAL(clicked()),this,SLOT(refreshLine()) );
23
24     p=new QPushButton("Beenden",this);
25     p->move(10,100);
26     connect(p,SIGNAL(clicked()),this,SLOT(close()) );
27 }
28
29 MyClass::~MyClass()
30 {
31 }
32
33 void MyClass::refreshLine()
34 {
35     setTextLine(edt->text());
36 }
37
38
39 int main(int argc,char**argv)
40 {
41     QApplication app(argc,argv);
42
43     MyClass *w=new MyClass;
44
45     w->show();
46
47     app.setMainWidget(w);
48
49     return app.exec();
50 }

```

Mit einem Aufruf von MOC erzeugt man aus der Header-Datei nun eine zusätzliche Datei, die ebenso wie `qtmoc.cpp` an den Compiler übergeben wird:

```

sh# $QTDIR/bin/moc -o moc_qtmoc.cpp qtmoc.h
sh# g++ -I$QTDIR/include -I/usr/X11/include -L$QTDIR/lib \
>> -L/usr/X11/lib -lqt -lX11 -lXext -lm qtmoc.cpp \
>> moc_qtmoc.cpp -o qtmoc

```

Wie man sieht, benötigt der Compiler Header-Dateien von Qt und X-Window und linkt auch gegen die entsprechenden Bibliotheken. Die Variable `$QTDIR` enthält hier den Pfad zu Qt.

Im Beispiel oben wird ein Klick auf den Button "Setzen" durch den Slot `refreshLine()` abgefangen. Dieser emittiert das Signal `setTextLine(const QString&)`, welches mit dem statischen Text verbunden ist und ihn neu setzt.



Abbildung 6.2: Qt Fenster mit Signalen und Slots

Kapitel 7

Internationalisierung

Internationalisierung bedeutet, dass Programme ihre Ausgaben in nahezu beliebigen (menschlichen) Sprachen präsentieren können. Dies ermöglicht es Personen, die nicht die Sprache des Autors sprechen, die Software zu nutzen. Im Falle von SNNS und Morgana hat es auch den Vorteil, dass man das Programm auf die Sprache einstellen kann, die man aus der lokalen Fachliteratur gewöhnt ist.

Internationalisierung steht immer im Zusammenhang mit Lokalisierung. Dabei ist Internationalisierung in diesem Kontext die Vorbereitung eines Programmes auf die Unterstützung mehrerer Sprachen. Lokalisierung ist die eigentliche Übersetzung der Texte des Programmes. Lokalisierung wird bei Open Source-Projekten normalerweise von größeren Übersetzungsteams vorgenommen, die die jeweilige Sprache beherrschen und projektunabhängig arbeiten.

7.1 Mögliche Lösungsansätze

Es werden verschiedene Ansätze zur Lösung dieser Aufgabe genutzt.

Direkte Übersetzung, also das direkte Editieren des Programmquelltextes wird heute kaum noch eingesetzt, da mit dieser Methode bei jedem Release die Übersetzung wieder von vorn beginnen müsste.

Feste Stringtabellen im Programm selbst sorgen dafür, dass das Programm mit einer Sprache kompiliert werden kann. Dabei werden die Texte meist über Zahlen indiziert. Diese Methode hat zwei entscheidende Nachteile:

1. es kann immer nur eine Sprache in einem Programm ausgeliefert werden (oder es müssen mehrere Versionen geliefert werden, die unterschiedliche Sprachen enthalten) und
2. die Indizierung über Zahlen ist zwar sehr effizient, bringt den Programmierer jedoch in die Lage einen Zusammenhang zwischen intuitionsfeindlichen Zahlen und einigen dutzend bis mehreren tausend Texten in Verbindung bringen zu müssen.

Trotzdem ist diese Methode zum Beispiel in der Windows-Welt sehr verbreitet.

Externe Tabellen ermöglichen es dagegen alle verfügbaren Sprachen zusammen mit einer Programmversion auszuliefern. Sie bieten außerdem den Vorteil, dass (zumindest theoretisch) die Sprache, während das Programm läuft, umgeschaltet werden kann. Auch hier gibt es wieder verschiedene Ansätze die Texte zu indizieren.

Zahlen als Index sind relativ effektiv, da sie sehr schnell verglichen werden können. Sie haben jedoch den Nachteil, dass unbedingt eine Sprache installiert und erreichbar sein muss. Catgets¹ löst dieses Problem indem in dem Aufruf, der den Text aus der Tabelle holt, zusätzlich ein Text in der Standardsprache des Programms übergeben wird. Nach wie vor bleibt hier das Problem, dass der Programmierer mit Zahlen umgehen muss, die zu verwalten nach einigen Versionen schwierig wird.

Text als Index ist die für Programmierer einfachste Lösung. Es wird einfach der Text in der Standardsprache des Programmes (meist Englisch) übergeben und die Internationalisierungs-Bibliothek versucht eine passende Übersetzung zu finden. Falls keine Übersetzung existiert kann bei dieser Methode der Index als Ausgabertext verwendet werden.

Ein klarer Vorteil ist, dass der organisatorische Aufwand für den/die Programmierer wesentlich geringer wird. Einziger Nachteil ist, dass Strings wesentlich aufwändiger zu vergleichen sind als Integer-Zahlen. Aber einerseits kann dies durch geeignete Wahl von Hashsummen beschleunigt werden und andererseits sind die heutigen Rechner schnell genug, um dies zu kompensieren, zumal Internationalisierung nahezu ausschließlich in Programmteilen vorkommt, in denen Geschwindigkeitsoptimierungen im Nanosekundenbereich nur eine untergeordnete Rolle spielen: der (grafischen) Oberfläche.

7.2 GNU gettext

Ich habe mich bei dem vorliegenden Programm für GNU gettext entschieden. Dieses Paket hat sich für freie Software zu einem Standard entwickelt, daher kann die Lokalisierung den eingespielten Übersetzerteams überlassen werden.

GNU gettext gehört zu den Bibliotheken, die externe Tabellen benutzen und den Standardtext als Index nutzen. Die Nachteile dieser Methode versucht GNU gettext durch geschickte Optimierungen zu kompensieren. So werden die Texte alphabetisch sortiert, um eine effektive binäre Suche durchführen zu können. Weiterhin werden Hash-Tabellen angelegt, die die Suche weiter beschleunigen können.

GNU gettext bietet jedoch momentan keine Möglichkeit Zahlendarstellungen, Zeiten und die Position von Währungssymbolen den lokalen Gegebenheiten anzupassen.

Weitere Informationen dazu finden sich in [MGTT].

7.3 Morgana und Internationalisierung

In Morgana sind die Schnittstellen für Internationalisierung vorbereitet, sie wird jedoch in dieser Phase des Projektes noch nicht aktiv realisiert.

Speziell bedeutet dies, dass in (nahezu) allen relevanten Teilen der GUI alle Zeichenketten, die ausgegeben werden, in das Makro `i18n(x)` eingeklammert sind. Momentan "expandiert" dieses Makro zu `(x)`, was in der Realität die Zeichenkette unverändert stehen läßt. Sobald Morgana über die Phase der Alpha-Tests hinaus ist kann dieses Makro durch diese Definition ersetzt werden:

```
#define i18n(x) gettext(x)
```

effektiv sorgt dies dann dafür, dass die notwendigen Ersetzungen von GNU gettext vorgenommen werden.

Das Problem der Anpassung von Zahlendarstellungen sollte bei Morgana dadurch zu lösen sein, dass vorhandener Quelltext aus dem KDE-Projekt benutzt wird, das auf die selbe Bibliothek (Qt) aufsetzt und dieses Problem bereits gelöst hat².

¹catgets wurde von der X/Open-Group entwickelt, jedoch noch nicht standardisiert.

²Abgesehen von der technischen Unbedenklichkeit sollte es hier auch keine rechtlichen Schwierigkeiten geben, da beide Projekte der selben *freien* Lizenz unterliegen.

Kapitel 8

GNU Autoconf

Da Morgana plattformunabhängig entwickelt werden soll, muss es sich vor der Compilierung bereits den Gegebenheiten der jeweiligen Umgebung anpassen. Grundsätzlich kann man dies erreichen, indem alle entsprechenden Sektionen jedes Quelltextes mit `#ifdef`-Anweisungen eingerahmt werden, die dynamisch auf das System reagieren. Komfortabler wird dies, indem diese Anweisungen in einer zentralen Header-Datei gesammelt werden, die dann von allen relevanten Quelldateien eingebunden wird. Diese Bedingungen decken jedoch nur einige der möglichen Gegebenheiten ab, die durch Preprozessor-Symbole abgetestet werden können. Es bietet sich also an ein Programm zu nutzen, das das System vor der Compilierung untersucht und alle relevanten Informationen sammelt und in einer Header-Datei ablegt und die Make-Skripte automatisch anpasst (z.B. Kommando des Compilers, Pfade der Bibliotheken). Solche Tools wären zum Beispiel `IMake`, das für jedes bekannte System eine Makrodatei besitzt, oder GNU Autoconf, das das System dynamisch nach jeder einzelnen Eigenschaft durchsucht. Bei Open Source Projekten scheint sich GNU Autoconf (siehe auch [MAC]) durchgesetzt zu haben, da es für Entwickler relativ einfach zu bedienen ist und nur Makros für alle benötigten Systemeigenschaften benötigt (die meisten davon sind schon mit Autoconf mitgeliefert) und sich dann dynamisch den Systemen anpasst — das ist besonders praktisch, wenn der Administrator Veränderungen am System vorgenommen hat.

Autoconf ist eine Sammlung von Makros in der Sprache `m4`, die es dem Entwickler ermöglichen ein Shell-Skript zu erstellen, das das installierte System nach benötigten Eigenschaften durchsucht. Besonders für Open Source Software ist Autoconf sehr beliebt, da es diese Aufgabe sehr effektiv zu bewältigen hilft.

8.1 Autoconf aus Nutzersicht

Ein Nutzer braucht lediglich das Skript `./configure` in dem Verzeichnis aufzurufen, in dem sich die Quelltexte eines Programmes befinden, um danach mit `make install` das Paket automatisch erstellen und installieren zu können. `./configure` versteht von sich aus einige Parameter mit denen man zum Beispiel den Pfad einstellen kann, in den das Paket installiert wird (`--prefix=pfad`).

Der Nutzer braucht also nicht von Hand die Makefile's an das eigene anpassen. Autoconf generiert automatisch geeignete Werte. Falls dies nicht möglich ist bekommt man meistens von diesem Skript angezeigt, was getan werden muss, um das Paket erfolgreich zu installieren.

8.2 Autoconf für kleine Projekte

In kleineren Projekten, die nur aus wenigen Dateien bestehen oder gegenüber dem System sehr anspruchslos sind, reichen die von GNU Autoconf mitgebrachten Makros vollkommen aus. Es existieren Tests für nahezu alle, aus der Sicht eines Programmierers, wichtigen Eigenschaften eines Unix-artigen Systems.

Input-Dateien für Autoconf haben normalerweise die Endung `.in`. Die zentrale Datei ist dabei `configure.in`, in dieser Datei stehen Makroaufrufe, die definieren, welche Tests durchgeführt werden sollen. Eine typische Datei wäre diese:

```
1 dnl #####
2 dnl AutoConf file
3 AC_INIT(main.cpp)
4
5 dnl #####
6 dnl Checks for programs.
7 CXXFLAGS="-O2 -Wall"
8 AC_LANG_CPLUSPLUS
9 AC_PROG_CXX
10 AC_PROG_INSTALL
11
12 dnl #####
13 dnl Checks for header files.
14 AC_CONFIG_HEADER(config.h)
15 AC_HEADER_STDC
16
17 dnl #####
18 dnl end phasis
19 AC_OUTPUT(Makefile)
20
21 dnl #####
22 dnl EOF
23 dnl #####
```

Die Makros, die von Autoconf mitgebracht werden, beginnen mit dem Kürzel `AC`. Andere Pakete, wie Automake, definieren ihre eigenen Kürzel, damit bei neuen Versionen die Makros immernoch unterscheidbar sind.

Diese Datei muss immer mit `AC_INIT` begonnen und mit `AC_OUTPUT` abgeschlossen werden. `AC_INIT` bekommt als Parameter eine Datei, an der es erkennt, ob es sich im richtigen Verzeichnis befindet. `AC_OUTPUT` werden alle Dateien übergeben, die erzeugt werden sollen, außer Header-Dateien, die durch `AC_CONFIG_HEADER` erzeugt werden. Kommentare werden durch `dnl` oder `#` eingeleitet. Im Wesentlichen handelt es sich bei allen anderen Makros um Tests. Diese überprüfen im obigen Beispiel das Vorhandensein eines C++-Compilers, einiger Header-Dateien und des Programmes `install`.

Damit das `configure`-Skript ein angepasstes Makefile erzeugen kann, muss eine Datei `Makefile.in` existieren. Diese Datei enthält überall dort, wo ein Wert von `configure` eingesetzt werden soll, eine Variable der Art `@VAR@`.

Mit einem einfachen Aufruf von `autoconf`; `autoheader` erzeugt man nun das Shell-Skript `configure` und die Header-Datei `config.h.in`. Der Nutzer ist ab diesem Moment von `autoconf` unabhängig.

8.3 Autoconf Erweiterungen

Man kann GNU Autoconf um eigene Makros erweitern. Dies ist insbesondere bei komplexen Projekten notwendig. Morgana nutzt diese Möglichkeit unter anderem, da es die Bibliothek Qt einbinden muss. Dazu definiert man eine Makro-Datei namens `aclocal.m4`.

Dieses Makro sucht nach Qt:

```
1 dnl
2 dnl local macros for Qt Apps
3 dnl
4 dnl written by Konrad Rosenbaum
5 dnl
6
7 AC_DEFUN(MORGANA_CHECK_QTDIR,[
8 AC_MSG_CHECKING(path to Qt)
9 AC_CACHE_VAL(morgana_cv_check_qtmdir,[
10   test -z $QTDIR && {
11   for dir in /lib/qt /usr/lib/qt /opt/qt /usr/local/qt ; do
12   test -d $dir && { morgana_cv_check_qtmdir=$dir ; break ; }
13   done
14   test -z $morgana_cv_check_qtmdir && {
15   echo "Error: couldn't find Qt, please set QTDIR and \
16   rerun configure"
17   exit 1
18   }
19   export morgana_cv_check_qtmdir
20   } || morgana_cv_check_qtmdir=$QTDIR
21 AC_MSG_RESULT($morgana_cv_check_qtmdir)
22 QTDIR=$morgana_cv_check_qtmdir
23 AC_SUBST(QTDIR)
24 ])
```

Das Makro wird durch `AC_DEFUN` definiert. Die anderen Makros sorgen dafür, dass entsprechende Nachrichten ausgegeben werden und die Werte in der Cache-Datei gespeichert werden können, um nicht bei jedem Aufruf das komplette Skript durchgehen zu müssen. Der größte Teil des Makros wird hier durch das eigentliche Shell-Skript eingenommen.

8.4 Morgana und Autoconf

Viele Projekte nutzen zusätzlich das Paket GNU Automake. Es bietet die Möglichkeit Makefiles im GNU-Stil automatisch erstellen zu lassen. Da Morgana noch in einem tiefen Entwickler-Stadium steckt habe ich mich vorerst gegen Automake entschieden, da ohne Automake die Makefiles wesentlich flexibler gestaltet werden können und kein Zwang besteht bestimmte Dateien zu erstellen.

Demzufolge benutzt Morgana auch nur einen Teil der Funktionalität von GNU Autoconf. So werden zum Beispiel keine Routinen zur Installation der Programme erzeugt. Vom Unterverzeichnis `morgana` aus kann man Morgana compilieren und starten:

```
./build
make start
```

`./build` startet automatisch die `configure`-Skripte von `SNNS` und `Morgana`. `Make` startet dann `Morgana`

mit einigen zusätzlichen Shell-Variablen, die momentan noch notwendig sind. Die Variable \$QTDIR sollte auf jeden Fall schon auf den Pfad zu einem Qt ab Version 2.x gesetzt sein.

Teil III

Design

Dieser Teil der Arbeit gibt einen kurzen Überblick über Teile des Morgana-Designs, die zu Beginn der Arbeit ausgearbeitet wurden und inzwischen teilweise umgesetzt sind (siehe Teil 12.3).

Kapitel 9

Konzepte

Beim Design der Applikation als Ganzes bieten sich im Fall eines Simulators, also auch Morgana/SNNS, zwei grundlegende Ansätze:

Monolithe¹ sind relativ einfach zu konzipieren und nutzen Uniprozessor-Maschinen optimal aus. Die Kommunikation zwischen den einzelnen Teilen des Systems ist sehr effektiv. Die Nachteile dieses Ansatzes sind jedoch, dass Multiprozessor-Maschinen nicht ausgenutzt werden und das Versagen einer Komponente auch zum Versagen aller anderen Komponenten führt.

Multithreading Anwendungen nutzen zwar Multiprozessor-Maschinen aus, haben aber ansonsten die selben Probleme wie Monolithe. Zusätzlich muss der Zugriff auf den gemeinsamen Speicher synchronisiert werden.

Unabhängige Prozesse müssen Informationen über serialisierte Kommunikationskanäle austauschen², d.h. die Kommunikation wird schwieriger und aufwändiger. Jedoch führt das Beenden eines Prozesses nicht zwangsläufig zum Beenden der anderen beteiligten Prozesse. Unabhängige Prozesse können theoretisch auch Workstation-Cluster ausnutzen, indem sie über das lokale Netzwerk verteilt werden. Ein derart flexible Lösung erfordert jedoch einen sehr hohen Design und Implementierungsaufwand, bietet aber auch sehr viele Möglichkeiten der Erweiterung — ich habe mich deswegen für diesen Weg entschieden.

9.1 Client/Server

Morgana ist in zwei nahezu unabhängige Programme unterteilt. Zum einen die graphische Oberfläche, die dafür sorgt, dass der Nutzer mit den neuronalen Netzen interagieren kann. Zum anderen der Server, der die eigentliche Funktionalität enthält.

Grafische Nutzerschnittstellen (GUI's) gelten allgemein als sehr fehleranfällig. Insbesondere haben sie oft die Eigenart das gesamte Programm zu beenden, wenn das grafische Subsystem aus irgendeinem Grund (sei es ein Programmfehler oder Fehlbedienung) beendet wird. Besonders bei sehr aufwändigen Kalkulationen kann dies sehr lästig werden, da diese dann von vorn begonnen werden müssen. Es liegt also nahe die Schnittstelle von der eigentlichen Funktionalität zu trennen und beide in unterschiedlichen Prozessen laufen zu lassen.

¹Erklärungen siehe Anhang B — Glossar

²auch bei Nutzung von "Shared Memory", da diese Prozesse oft unterschiedliche Speicheradressen nutzen

Dieses Vorgehen hat einige weitere Vorteile:

- Das System kann zumindest geringfügigen Nutzen aus Multiprozessor-Maschinen ziehen, da die grafische Oberfläche nicht den selben Prozessor belastet, wie die Berechnungen.
- Theoretisch können mehrere GUI's auf die selben Daten zugreifen und umgekehrt eine GUI mehrere Netzwerke gleichzeitig verwalten. Es sind also beliebige M:N-Beziehungen zwischen Clients (GUI's) und Servern denkbar.
- Bei entsprechender Implementation können diese Prozesse auch über ein lokales Netzwerk verteilt werden.
- Bei sehr langwierigen Simulationen kann die GUI beendet werden, während die Simulation im Hintergrund weiterläuft. Der Nutzer kann sich dann später über den Stand der Simulation informieren.

Ein deutlicher Nachteil dieses Vorgehens ist jedoch, dass eine zusätzliche Kommunikationsschicht zwischen GUI und Funktionalität geschoben werden muss. Dies bedingt einige zusätzliche Kopiervorgänge bei denen Daten über potentiell langsame Netzwerkschnittstellen transferiert werden müssen. Dieser Nachteil läßt sich nur durch sehr aufwändige und aggressive Optimierungen an der Kommunikationsschicht zu einem gewissen Grad kompensieren. Es muss bei der Implementation also darauf geachtet werden, dass so effektiv wie möglich kommuniziert wird.

9.2 Das Session-Konzept

Die Morgana-GUI besitzt die Möglichkeit mehrere sog. Sessions gleichzeitig zu verwalten. Jede Session repräsentiert dabei eine unabhängige Simulator-Instanz. Sie hat also eigene neuronale Netzwerke mit eigenen Parametern und wird dementsprechend in eigenständigen Dialogen dargestellt. Sessions sind durch eine eindeutige Beziehung zwischen Server (Simulator), Client (GUI) und einer Verbindung gekennzeichnet. Es können also durchaus mehrere Sessions zwischen dem selben Client und Server existieren, auch wenn dies in der Praxis nur selten sinnvoll sein dürfte. Da für alle bekannten Betriebssysteme Kommunikationsverbindungen eindeutige "Codes" besitzen, reicht in der Regel schon dieser "Code" aus, um eine Session zu identifizieren.

Dementsprechend sind die Session-ID's zu verstehen, die von Morgana benutzt werden. Sie bestehen aus einem Verbindungstyp (z.B. LHOST für TCP/IP-Verbindungen auf dem selben Rechner) und einem eindeutigen Code (bei TCP/IP die Port-Nummer).

Kapitel 10

Oberfläche

Ausgangspunkt des GUI-Designs war die vorhandene Oberfläche von SNNS, wobei die unangenehmen Eigenschaften Schritt für Schritt beseitigt werden sollten. Da in der Kürze der Zeit nur ein Teil dieser Oberfläche implementiert werden konnte, sind einige der Abbildungen hier nur Konstrukte, die mit wenigen Code-Zeilen und dem Programm GIMP¹ erstellt wurden.

Generell können viele der von der SNNS-xgui auf Buttons gelegten Funktionen in ein Menüsystem sortiert werden. Mit Hilfe dieser Menüs lassen sich auch weitere Funktionen verankern, die sonst zu viel Platz im Fenster verbrauchen würden, so zum Beispiel ein ausgefeiltes Hilfesystem oder Verbindungen zu anderen Fenstern, die bei Auswahl des jeweiligen Menüpunktes geöffnet oder in den Vordergrund geholt werden.

10.1 ControlCenter


Das “snns-control”-Fenster von SNNS erschließt sich für den Einsteiger erst nach sehr gründlichem Studium der SNNS-Dokumentation ([SNNS], Kapitel 4.3.3). Dies liegt einerseits daran, dass nur drei Widget-Typen benutzt werden: Label, Eingabezeile und Button. Alle zentralen Daten werden hier in einer einzigen Arbeitsfläche eines Fensters bearbeitet und ebenso alle Aktionen auf dem Netzwerk werden von hier gestartet und beendet.

Zunächst sind also die recht einseitigen Widgets durch passendere Widgets ihrer jeweiligen Aufgabe anzupassen. So müssten zum Beispiel die vier “Sel. Func”-Buttons durch ComboBoxes² ersetzt werden.

Danach sollten die einzelnen Aufgabengebiete des ControlCenters auf eigene SubDialoge aufgeteilt werden. Dies kann man sehr einfach mit dem TabDialog-Widget erreichen, indem die jeweiligen Aufgabengebiete auf eigene “Karteikarten” verteilt werden. Dadurch werden unabhängige Einstellungen auch sichtbar voneinander getrennt. Wie in [TAO-HCI] beschrieben handelt es sich dabei um einen iterativen Prozess, für den Designer des Interfaces ist es nur schwer möglich abzuschätzen, ob sein Design von den Nutzern angenommen wird. Es bleibt also abzuwarten, ob die gewählte Aufteilung wirklich gut genug ist. Qt erleichtert an dieser Stelle allerdings das Experimentieren, da die Aktionen im Hintergrund durch virtuelle Signal-Slot-Beziehungen und nicht durch die logische Struktur des Fensters bestimmt werden.

Das Ergebnis dieser Überlegungen zeigt Abbildung 10.1. Dabei handelt es sich jedoch noch um einen frühen Prototypen, der noch durch die Interaktion mit Nutzern ausgebaut und verfeinert werden muss.

¹ GNU Image Manipulation Program, ein freies Pixel-Grafik-Programm

² eine ComboBox sieht in etwa so aus: , sie bietet dem Nutzer die Wahl aus mehreren Alternativen, in diesem Fall jedoch ohne die Option eigene Alternativen hinzuzufügen.

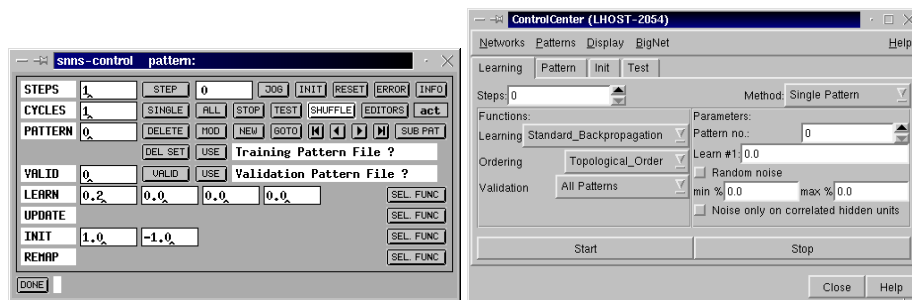


Abbildung 10.1: ControlCenter (links: SNNS, rechts: Morgana)

10.2 Display

Das 2D-Display von SNNS erzeugt eine relativ übersichtliche Darstellung neuronaler Netze. Die Manipulation dieser Netze wird allerdings stark erschwert. So sind einzelne Funktionen oft erst in der zweiten Hierarchie eines Menüsystems zu erreichen, das durch drücken der Strg-Taste und der rechten Maustaste stufenweise durch die Hierarchieebenen navigiert wird. Verschiebungen von Units werden nicht relativ zum Beginn der Mausbewegung sondern relativ zur linken oberen Ecke des Displays vorgenommen, dadurch verschwinden Units oft außerhalb der Darstellungsfläche, wenn man an der falschen Stelle ansetzt.

Die Lösung dieser Probleme ist mit Qt relativ einfach. Die Klasse QPopupMenu bietet die Möglichkeit kontextsensitive Menüs zu erstellen, die wie von anderen modernen Programmen gewohnt mit der rechten Maustaste geöffnet und direkt durchsucht werden können. Schwierig ist für das Programm lediglich die Entscheidung zwischen mehreren unterschiedlichen Popup-Menüs, dazu muss jeweils ausgewertet werden, welchem Element im Netzwerk der Mauszeiger am nächsten ist. Mit dem Konzept der Werkzeugleiste (Toolbar)³ können die wichtigsten Werkzeuge direkt ausgewählt werden.

Theoretisch unterscheiden sich 2D- und 3D-Display lediglich in der Darstellung des selben Netzwerkes. Beim 2D-Display werden lediglich die Z-Koordinaten bei der Projektion des Netzes weggelassen, bei der 3D-Darstellung wird ein kompletter 3D-Raum errechnet und dann auf ein "Leinwand-Widget" projiziert. Man kann also theoretisch beliebig zwischen 2D- und 3D-Darstellung umschalten, während die Datenstrukturen im Hintergrund die selben bleiben (weitere Details dazu in Kapitel 15.4, Seite 55).

10.3 Pattern Editor

Während man sich in der SNNS-xgui per Buttons durch die einzelnen Muster eines Pattern-File navigieren muss, sollte es eigentlich möglich sein alle vorhandenen Muster in einer tabellarischen Darstellung zu sehen und mit einem Mausklick auf ein Muster die Möglichkeit zu bekommen dieses Muster zu verändern. Abbildung 10.3 zeigt einen ersten Ansatz dazu.

In diesem Entwurf fehlen jedoch noch Möglichkeiten diese Manipulation direkt im 2D-/3D-Display vorzunehmen und auszuwählen, ob ein Muster zum Trainieren, Validieren und/oder Testen des Netzes verwendet werden soll. Außerdem wäre es denkbar Muster in einer Hierarchie zu ordnen oder Prioritäten festzulegen.

³in Abbildung 10.2 der Balken mit den Icons, unter dem Menübalken

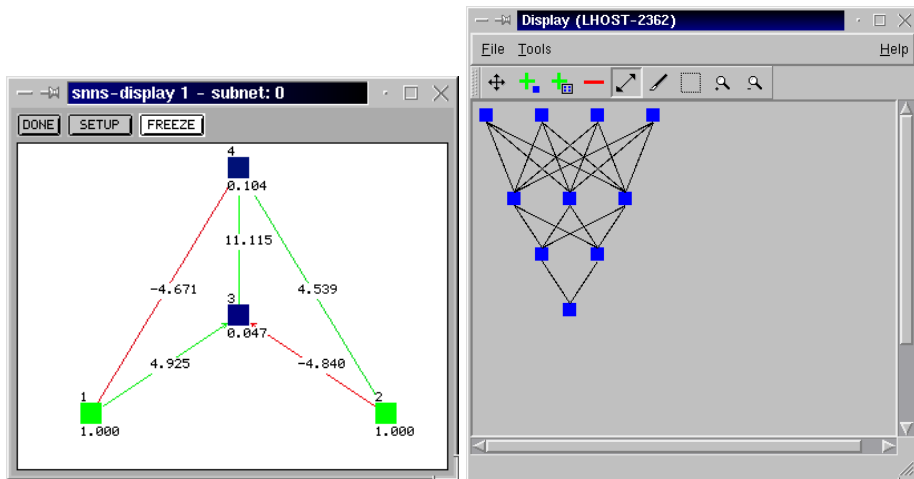


Abbildung 10.2: 2D-Display (links: SNNS, rechts: Morgana)

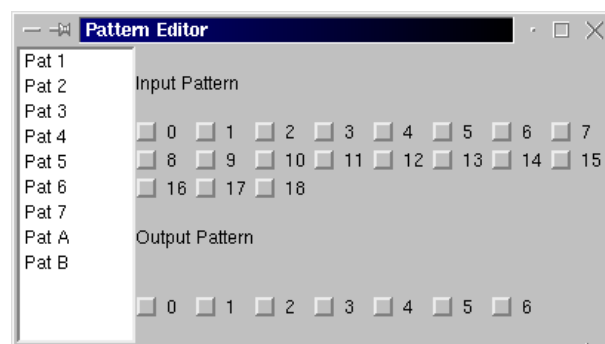


Abbildung 10.3: Entwurf eines Pattern Editor

10.4 Diagramme

Die SNNS-xgui kennt mehrere unterschiedliche Typen von Diagrammen: Graph (Verlauf der Time-Error-Kurve), Analyzer (X-Y, Zeit-Y, Zeit-Error Kurven), Weight (farbliche Darstellung der Gewichtsmatrix) und Projection (Analyse zwischen zwei Input- und einem Output-Wert). Diese Diagrammtypen sollten in einem einzelnen konfigurierbaren Fenster darstellbar sein. Jede Dimension eines Diagramms kann dann mit beliebigen Werten des Netzwerkes verknüpft werden. Dazu müssen einige generische Diagrammklassen definiert werden:

2D-Diagramm: auf einer zweidimensionalen Projektionsfläche wird der Zusammenhang zwischen zwei Werten durch einzelne oder miteinander verbundene Punkte dargestellt.

2D+thickness: wie 2D, aber eine dritte Dimension wird durch die Dicke der Punkte dargestellt.

2D+color: wie 2D, aber eine dritte Dimension wird durch Farbe dargestellt.

2D+shape: wie 2D, aber die Punkte erhalten eine Form für diskrete Werte.

2D+XX: die Modi "thickness", "color" und "shape" könnten gemischt werden.

3D: drei Werte werden wie im 3D-Display räumlich projiziert.

3D+XX: die Modi "thickness", "color" und "shape" könnten auch hier verwendet werden.

Distance: speziell für SOM-Netze eine Darstellung der Distanz zwischen mehreren Neuronen in einem ein-, zwei- oder dreidimensionalen Raum. Einige zusätzliche Werte könnten hier mit Farbe dargestellt werden (z.B. die Aktivierung des jeweiligen Neurons).

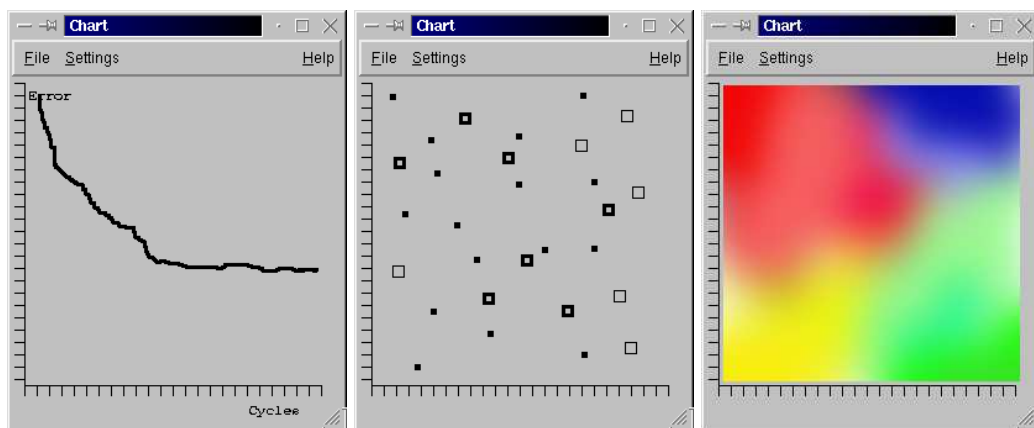


Abbildung 10.4: Entwurf eines Diagramm-Fensters (von links: 2D (Error-Graph), 2D+shape, 2D+color)

10.5 Online-Hilfe

Da nahezu jedes Fenster ein eigenes Menü enthält, können die einzelnen Fenster individuell mit einer Hilfefunktion versehen werden. Qt bietet Klassen, mit denen HTML-Texte dargestellt werden können. Damit ist es möglich auch Grafiken, Tabellen und andere Gestaltungselemente in die Online-Hilfe einzufügen und die Hilfeseiten mit einem beliebigen ASCII- oder HTML-Editor zu verfassen.

Kapitel 11

Inter-Process-Communication

Die Abkürzung IPC steht für “Inter-Process-Communication”. Es bezeichnet Schnittstellen, die der Kommunikation zwischen Prozessen dienen, sowohl auf dem selben Rechner, als auch über Netzwerke.

Die klassischen Formen des IPC nutzen Pipes, Sockets, Semaphoren, Message-Queues und Shared-Memory.

Neuere Bibliotheken setzen auf diesen Formen des IPC auf und implementieren höhere Abstraktionsebenen. So bieten die Pakete MPI und PVM netzwerk-transparent die Möglichkeit Prozesse zu kontrollieren und Nachrichten auszutauschen. Objektorientierte Tools wie CORBA (entwickelt von der OMG), COM (MicroSoft) oder DCOP (Entwicklerteam von KDE2) bieten Möglichkeiten Objekte netzwerk-transparent abzubilden und aufzurufen.

Diese existierenden Bibliotheken haben jedoch alle Eigenschaften, die dazu führten, dass sie nicht direkt in Morgana eingesetzt werden können:

MPI und PVM liegen auf einer zu niedrigen Abstraktionsebene, um objektorientierte Kommunikation abzubilden, und PVM benötigt zusätzlichen Administrationsaufwand, der dem Nutzer nicht zugemutet werden soll.

CORBA benötigt eine zusätzliche Sprache (IDL, Interface Definition Language), um die Interfaces zwischen den beiden Prozessen zu definieren. Das KDE2-Projekt hatte ursprünglich geplant CORBA einzusetzen, die Erfahrung zeigte jedoch, dass IDL die Entwickler abgeschreckt hatte.

COM ist ein Windows-spezifisches Protokoll, das auf Unix'en nicht/kaum verfügbar ist.

DCOP ist der Ersatz des KDE2-Teams für CORBA, es ist theoretisch ideal, da es sehr kompakt und effektiv ist und die Eigenschaften von Qt ausnutzt, es ist jedoch an das X11-Protokoll gebunden und eignet sich damit nicht für die Kommunikation mit Prozessen ohne GUI und solchen, die auf Windows laufen.

Der hier vorgestellte Entwurf der Morgana-eigenen IPC-Schicht ist als Prototyp zu verstehen, der lediglich zeigen soll, ob und unter welchen Bedingungen eine Kommunikation zwischen den beiden Morgana-Prozessen effektiv möglich ist. Er orientiert sich noch an klassischen IPC-Formen mit Pipes und Message-Queues. Eine Weiterentwicklung dieses Designs in Richtung DCOP ist in Kapitel 16 beschrieben.

11.1 Konzept

Morgana-IPC übermittelt Kommunikationspakete (Klasse `IPackage`), die jedes für sich eine Nachricht darstellen. Damit zwei Prozesse kommunizieren können, muss eine Verbindung (`IConnection`) aufgebaut werden.

Morgana-IPC implementiert asynchrone, ereignisgesteuerte Kommunikation. Das heißt auf eine Nachricht muss nicht unbedingt eine Antwort erfolgen oder es können mehrere andere Nachrichten übermittelt werden, bevor die Antwort auf eine Nachricht übermittelt wird. Das darüber liegende Programm wird über Qt-Signale aufgefordert eingegangene Nachrichten zu verarbeiten. Um Daten auf den Kommunikationskanälen zu empfangen, wird das Scheduling von Qt benutzt. Im Falle des Servers wird `QApplication` ein weiterer Parameter übergeben, der die grafische Umgebung abschaltet.

Kapitel 12

Server

Der SNNS-Server ist als Schnittstelle zwischen dem SNNS-Kern und dem Morgana-IPC konzipiert. Er übersetzt also zwischen den Welten des objektorientierten Morgana und des funktions-/datenorientierten SNNS-Kernels.

Hier werden die Ideen dargestellt, die hinter der momentanen Implementation des SNNS-Servers stehen. Wie in Abschnitt 14.4 dargestellt haben diese sich jedoch als sehr aufwändig zu implementieren herausgestellt. Ein erweitertes und verbessertes Design wird in Kapitel 17 dargestellt.

12.1 Kommunikation

Das Morgana-eigene IPC und einige zusätzliche Funktionen im SNNS-Server bilden einen äußeren Ring um den eigentlichen Kernel und sichern ab, dass kommuniziert werden kann, indem ein IPC-Server erzeugt, Verbindungen und Pakete entgegen genommen und Authentifikation durchgeführt werden. Diese Schicht wird direkt vom Qt-eigenen Scheduler (also einem QApplication-Objekt) kontrolliert. Dieser Teil des Servers muss also lediglich die IPC-Objekte koordinieren und die Pakete korrekt zuordnen.

12.2 Zugriff

Der Zugriff auf den SNNS-Kern erfolgt direkt unterhalb der Kommunikationsschicht. Sobald die Kommunikationspakete den Zugriffsmethoden zugeordnet sind, werden sie von diesen interpretiert und entsprechende Antworten an die Kommunikationsschicht zurückgegeben.

Bei bestimmten Operationen (z.B. Löschen einzelner Neuronen) müssen auch zusätzliche Informationen an alle verbundenen Clients geschickt werden, da sich dadurch einige Daten/Adressen im SNNS-Kern verschieben.

12.3 Ergänzungen

SNNS speichert lediglich das eigentliche Netzwerk im Kernel ab. Metadaten, wie der Typ des gespeicherten Netzwerkes gehen dabei verloren. Diese Metadaten müssten vom SNNS-Server selbst gespeichert und den entsprechenden Daten im SNNS-Kernel zugeordnet werden.

Teil IV

Implementierung

Dieser Teil beschreibt die momentane Implementierung der einzelnen Bestandteile von Morgana. Die Implementation erfolgte innerhalb der ersten vier von den fünf Monaten, die diese Arbeit in Anspruch nahm.

Da für ein Projekt dieser Größenordnung vier Monate nur ein verschwindend geringer Teil sind, können die entstandenen Quelltexte lediglich als Grundlagen der Projektinfrastruktur, Vorstudien und Prototypen betrachtet werden. Am Ende der jeweiligen Kapitel werden daher auch die aufgetretenen Probleme und noch fehlenden Eigenschaften näher betrachtet. Mögliche Lösungen dafür finden sich in Teil 15.7.

Kapitel 13

Inter-Process-Communication

13.1 Implementierung

Das Morgana-IPC besteht aus drei wesentlichen Arten von Klassen: Server, Kommunikation und Nachrichten, die hier kurz dargestellt werden. Genauer zu den jeweiligen Methoden kann der Klassendokumentation des IPC entnommen werden. (Auf der CD: /dipl/morgana/doc/classes/ipc)

13.1.1 Der Server

`IListener` repräsentiert einen Server, der Verbindungen entgegennimmt. `IListener` wirft dabei das Signal `newConnection(IConnection*)` aus. Dadurch kann man die neue Verbindung in einem eigenen Slot entgegennehmen und die Signale der Verbindung mit den weiteren Slots verbinden.

Dazu wird von der Klasse `IListener` eine TCP/IP-Socket geöffnet und per `bind` und `listen` als Server-Socket registriert. Per `select` kann nun auf eingehende Verbindungen gewartet werden (als Signal wird hier ein `read`-Ereignis benutzt). Per `accept` kann dann eine neue Socket erzeugt werden, die die neue Verbindung repräsentiert. `select` wird hier durch ein Objekt der Klasse `QSocketNotifier` im Qt-internen Scheduler durchgeführt.

13.1.2 Kommunikation

`IConnection` repräsentiert eine Verbindung über die Nachrichten geschickt (Slot `send(IPackage&)`) und empfangen (Signal `receive(IPackage&)`) werden können.

Ein Objekt der Klasse `IConnection` kann sowohl vom Nutzer (unter Angabe eines Servers), als auch von `IListener` erzeugt werden. Es stellt in jedem Fall eine Hülle um eine existierende Socket dar.

13.1.3 Nachrichten

Nachrichten werden momentan nur durch die Klasse `IPackage` repräsentiert. Diese Klasse kann nahezu wie eine Datenstrom behandelt werden, indem sich Werte mit dem Operator `<<` angefügt und dem Operator `>>` ausgelesen werden können.

Jedes Paket transportiert dabei einen ID-String und binäre Daten. Der ID-String identifiziert das Paket, was es den Kommunikationspartnern ermöglicht, das Paket einer Objekt-Methode oder einer Funktion zuzuordnen, die dieses Paket bearbeiten kann. Der Typ der Daten wird nicht gespeichert. Die beiden Programme müssen sich also selbst um die korrekte Interpretation der Daten kümmern.

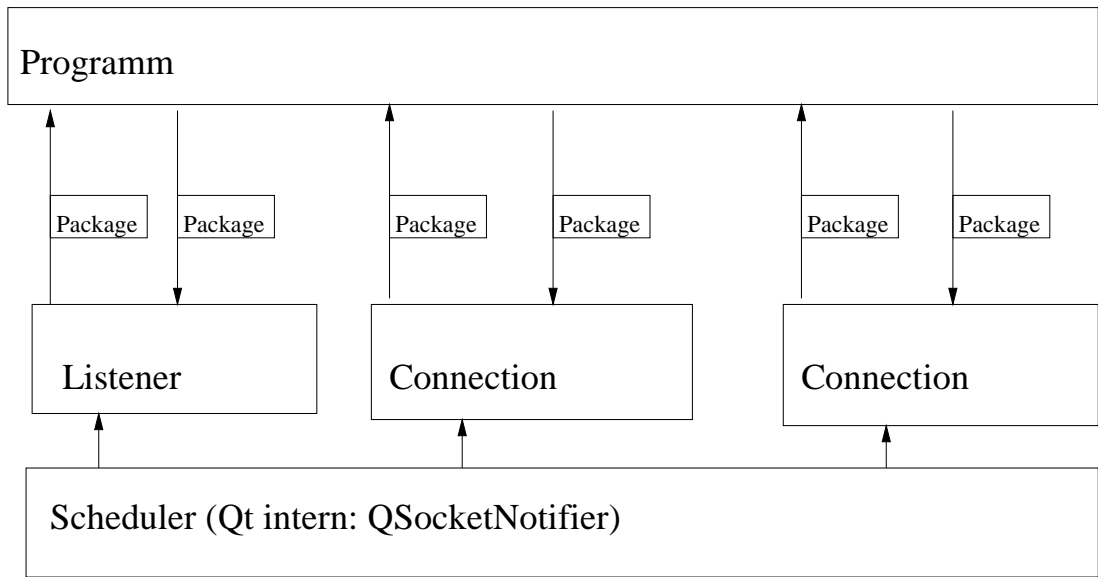


Abbildung 13.1: Beziehungen zwischen den IPC-Klassen

13.2 Probleme

In der momentanen Implementierung muss jedes Ereignis in ein Paket gewandelt (also serialisiert) werden. Dies stellt in einem objektorientierten Programm einen zusätzlichen Aufwand dar. Einfacher wäre es, wenn der Programmierer kaum merken würde, dass er den momentanen Prozess verlässt und eine Anfrage in einem anderen Prozess bearbeitet wird. Eine Erweiterung in diese Richtung wird in Kapitel 16 (Seite 64) dargestellt.

Kapitel 14

SNNS-Server

Die momentane Implementation des SNNS-Servers enthält nur einige prototypische Programmteile, die zeigen sollen, ob und wie eine Abstraktion des SNNS-Kernels in Richtung Objektorientierung funktionieren kann. Dementsprechend kann diese Implementierung nur als Test der Prinzipien hinter dem Morgana-IPC und dem SNNS-Kern sowie deren Kopplung betrachtet werden.

14.1 Konzept

Die Kommunikation zwischen Client (GUI) und Server erfolgt über Datenpakete. Die Pakete, die der Client an den Server sendet, enthalten jeweils die Identifikation einer auszuführenden Funktion und die dazu notwendigen Daten. Die Antworten enthalten wieder eine Identifikation und Daten, die der Client in eine Darstellung umsetzen kann.

Da SNNS selbst prozedural aufgebaut ist, ist es relativ schwierig eine objektorientierte Abstraktion auf den Kernel aufzusetzen. Als Prototyp für eine spätere Implementation wird in diesem Stadium von Morgana lediglich eine sehr schwach objektorientierte Schnittstelle implementiert. Dabei wird der gesamte SNNS-Server als ein großes Objekt betrachtet, das in der Lage ist über kleinere Objekte (Pakete) mit dem Client zu kommunizieren.

Ein verbessertes Design ist in Kapitel 17 beschrieben.

14.2 Paket-Format

Jedes Paket enthält eine Identifizierung, die angibt welcher "Aufgaben-Domäne" es angehört. Die Domänen reichen von protokollarischen Paketen (z.B. `login` und `admin`) über die eigentliche Arbeit mit den Netzwerken (`get`, `set` und `learn`) bis zu Workarounds für Inkompatibilitäten zwischen den Konzepten von Morgana und SNNS (SNNS).

Da eine Domäne noch nicht vollständig aussagt welche Aufgabe ein Paket hat, wird als erster Parameter ein String übergeben, der gegenüber Client und Server eindeutig identifiziert, wie dieses Paket interpretiert werden muss. In der Regel wird sowohl für die Anfrage, als auch die Antwort die selbe Identifikation genutzt, Server und Client können Pakete mit gleicher Identifikation durchaus unterschiedlich interpretieren.

Für das Paket "unit" aus der Domäne "get" sieht die Anfrage des Clients so aus:

IPackage("get")	
QString pid="unit"	
uint32 net	Nummer des Subnetzes
uint32 unit	Nummer der Unit

Die Antwort des Servers ist schon etwas komplizierter:

IPackage("get")	
QString pid="unit"	
uint32 net	Nummer des Subnetzes
uint16 num	Anzahl der Units
pro unit sind dann diese Daten enthalten:	
uint16 flags	sagt aus, welche Daten transportiert wurden, da nicht immer alle Felder belegt werden
QString proto	Prototyp-Name der Unit
sint32 posx, posy, posz	XYZ-Koordinaten der Unit
float bias	Bias-Wert des Neurons
float iact	initial activation
QString actf	Aktivierungsfunktion
QString outf	Output-Funktion
QString name	Name des Neurons
float curract	aktuelle Aktivierung
uint32 id	Nummer der Unit
QString err	error-string, falls bei der Interpretation ein Fehler auftrat

14.3 Umsetzung im Server

Der Server betrachtet jedes Paket als ein Ereignis, auf das er eine Reaktion finden muss. In einer zentralen Methode werden alle ankommenden Pakete nach Domänen sortiert und an andere Methoden weitergereicht¹:

```
void SMain::request(IPackage&p, IConnection*c)
{
    QString name=p.name();
    if(sl->findIndex(name)<0)return;
    //find package handler
    if(name=="request")reqRequest(p,c);else
    if(name=="set")reqSet(p,c);else

    if(name=="login")reqLogin(p,c);else
    if(name=="SIGNAL")reqSignal(p,c);else
    if(name=="chat")reqChat(p,c);
}
```

Nach diesem ersten Test wird das Paket (p) an eine Domänen-Methode weitergereicht. Diese entscheidet dann über die weitere Verarbeitung des Paketes:

¹Hinweis: der abgedruckte Quelltext entspricht nur einem Ausschnitt der eigentlichen Funktion, er soll hier aber genügen, da der Rest "lediglich" zum Debuggen und zur Zugriffssteuerung da ist.

```

void SMain::reqGet(IPackage&p, IConnection*c)
{
    p.pos(0);
    QString s;
    p>>s;

    if(s=="unit")getUnit(p,c);else
    if(s=="network")getNet(p,c);else
    if(s=="full")getFull(p,c);else
    if(s=="area")getArea(p,c);else
    if(s=="numbers")getNumbers(p,c);
}

```

In dieser Methode wiederum wird ausgewertet, welche spezielle Aufgabe das Paket zu erfüllen hat. Diese wird dann von einer Funktion ausgeführt, die die Kommunikation mit dem eigentlichen SNNS-Kern übernimmt und die, in ein neues Paket eingepackte, Antwort an den Client zurückschickt (über `c->send(...)`):

```

static void getUnit(IPackage&p, IConnection*c)
{
    uint32 sub,num;
    p>>sub;
    p>>num;
    //test for validity of request:
    if(num==0 || num>krui_getNoOfUnits()){
        //unit doesn't exist: inform client to delete it
        IPackage op("set");
        op<<"delete unit"<<sub<<(uint16)1<<num;
        c->send(op);
        return;
    }
    //prepare answer package:
    IPackage op("get");
    op<<QString("unit")<<sub<<(uint16)1;
    SUnit u;
    u.mask=UNUM|UPOS|UBIAS|UIACT|UACT|UACTF|UOUTF|UNAME;
    u.id=num;
    /*...
    Access on SNNS to fill u
    ...*/
    setUnit(u,op);
    c->send(op);
}

```

14.4 Probleme

Wie schon in Abschnitt 13.2 erwähnt ist das Paketformat des momentanen Morgana-IPC kaum geeignet eine objektorientierte Kommunikation zu ermöglichen. Das Problem besteht darin, dass die empfangenen Daten zunächst ausgepackt und in eigene Strukturen gespeichert werden müssen und nach der Verarbeitung wieder serialisiert und auf dem selben Weg zurückgeschickt werden müssen.

Wünschenswert wäre eine Methode virtuelle Strukturen zur Verfügung zu stellen, deren Elemente sich den Gegebenheiten anpassen und typisieren lassen. Wenn das Morgana-IPC diese Strukturen dann direkt aufnehmen und selbständig serialisieren kann, wird die Kommunikation erheblich erleichtert. Noch eleganter wären Signal-Slot-Beziehungen über Prozessgrenzen hinweg.

Weiterhin als sehr problematisch haben sich einige "Features" des SNNS-Kernels selbst herausgestellt. So werden beim Löschen einzelner Units alle darauffolgenden Units so weit verschoben, dass die Adresslücke aufgefüllt wird, womit sich im gesamten System die ID's der Units verschieben. Damit müssen die Daten auf der Client-Seite diesen neuen Gegebenheiten angepasst werden.

Kapitel 15

Oberfläche

Die Implementation der grafischen Oberfläche ist am weitesten fortgeschritten und hat bisher fast die gesamte Implementationszeit in Anspruch genommen.

Die Quelltexte befinden sich auf der CD im Verzeichnis `dipl/morgana/gui`. Die Dateien tragen dabei jeweils den Namen der darin implementierten Klasse in Kleinschreibung und ohne das vorangestellte "M". Nur sehr kleine Klassen oder Hilfsklassen sind in einer Datei zusammengefasst oder in die Dateien der jeweiligen Hauptklasse mit eingefügt.

15.1 Architektur

Die zentrale Instanz der GUI ist die Klasse `MSession`. Diese Klasse setzt direkt auf der IPC-Schicht auf und verbindet damit die GUI mit dem eigentlichen Simulations-Server.

Die jeweiligen GUI-Objekte können sich von dieser Klasse jeweils Pakete mit einer bestimmten ID liefern lassen. So sieht das in `chat.cpp` aus:

```
s->connectPackage( " chat " , this , SLOT( receive( IPackage& ) ) );
```

mit dieser Anweisung wird die Session (`s`) angewiesen alle ankommenden Pakete mit der ID "chat" an das Chat-Fenster¹ (hier: `this`) auszuliefern.

Umgekehrt werden Pakete über `MSession::send(IPackage&)` abgeschickt.

15.2 Hauptfenster und Sessions

Das Hauptfenster (siehe Abbildung 15.1) übernimmt die Steuerung über die Session-Objekte. Von hier aus können neue Sessions gestartet werden, laufende Sessions beendet und zukünftig auch konfiguriert werden.

Sessions sind von der Klasse `QWidget` abgeleitet, das bedeutet sie sind darstellbar. Die Klasse `MSession` hat eine ambivalente Funktion in Morgana, einerseits übernimmt sie die Darstellung einer Verbindung zu einem Server, andererseits kontrolliert sie aber auch diese Verbindung. Dies stellt insofern kein Problem dar, als dass die Session-Objekte selbst relativ einfach aufgebaut sind und in das einzige permanent aktive Fenster (das Hauptfenster) einbezogen sind. Damit werden die Sessions sauber beendet, wenn das Hauptfenster geschlossen wird (was ja bedeutet, dass auch alle anderen Fenster entweder schon geschlossen sind oder jetzt geschlossen werden).

¹Die Chat-Funktion war ein Test, ob die Pakete überhaupt korrekt ankommen, daher ist es auch am einfachsten gehalten.

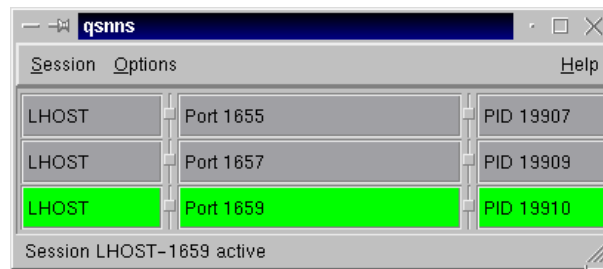


Abbildung 15.1: Hauptfenster

Session-Objekte sind intern über eine doppelt-ringförmig verkettete Liste verbunden. Dadurch kann aus der Liste der vorhandenen Sessions immer die herausgesucht werden, auf die sich die Kommandos des Hauptfensters beziehen. Eine der Sessions ist dazu als “aktiv” markiert, was in der Implementierung bedeutet, dass ein statischer Pointer (`static MSession*first;`) auf diese Session zeigt. Wird die aktive Session beendet wird aus dem Ring der anderen Sessions die nächste ausgewählt und aktiviert.

Um von der Session empfangene Pakete zu erhalten, muss sich ein Objekt mit der Methode `connectPackage` registrieren. Um diese Registrierung abzumelden wird die Methode `disconnectPackage` verwendet. Die Auslieferung der Pakete erfolgt dann im Hintergrund über Objekte der Klasse `MSessionDeliver`.

15.3 ControlCenter

Das ControlCenter von Morgana entspricht von der (geplanten) Funktion her dem SNNS-control-Fenster der `xgui`. Wie bereits in Abschnitt 10.1 erwähnt, wurden die aktiven Komponenten des ControlCenters neu geordnet und für die jeweilige Aufgabe passendere Widgets ausgesucht (siehe auch Abbildung 10.1 auf Seite 37). Da dieser Dialog noch keine Funktionalität implementiert hat, sieht die Klasse `MControlCenter` noch relativ einfach aus.

Bisher beschränkt sich diese Klasse darauf, ein `QTabWidget` als zentrales Element aufzubauen und über die Methoden `fillTabLearning`, `fillTabPattern`, `fillTabInit` und `fillTabTest` zu füllen. Lediglich einige der Menüeinträge sind bereits aktiv (`Network`→`Close` und `Display`→`2D Display` sowie die Einträge in `Help`).

15.4 Display

Das Display ist mit Abstand der komplexeste Teil der GUI. Es hat sehr große und komplexe Daten zu speichern und darzustellen. Das Display hat dabei einen Kompromiss zwischen möglichst wenig verbrauchtem Speicher und flüssiger Darstellung zu finden. Ersteres erfordert die Daten im Server zu belassen und nur bei Bedarf anzufordern, letzteres eine möglichst komplette Pufferung all jener Daten, die dargestellt werden können. Hier bestehen also noch weitreichende Möglichkeiten zur Optimierung. Wo das Optimum liegt wird sich wahrscheinlich erst bei längerer Benutzung zeigen.

Das Display kann als für Morgana so zentral betrachtet werden, wie die Speicherverwaltung für ein Betriebssystem. Das Tuning dürfte demnach ähnlich schwierig werden. Die momentane Implementierung besitzt schon einige Anlagen zur Optimierung, ist selbst aber noch völlig unoptimiert.

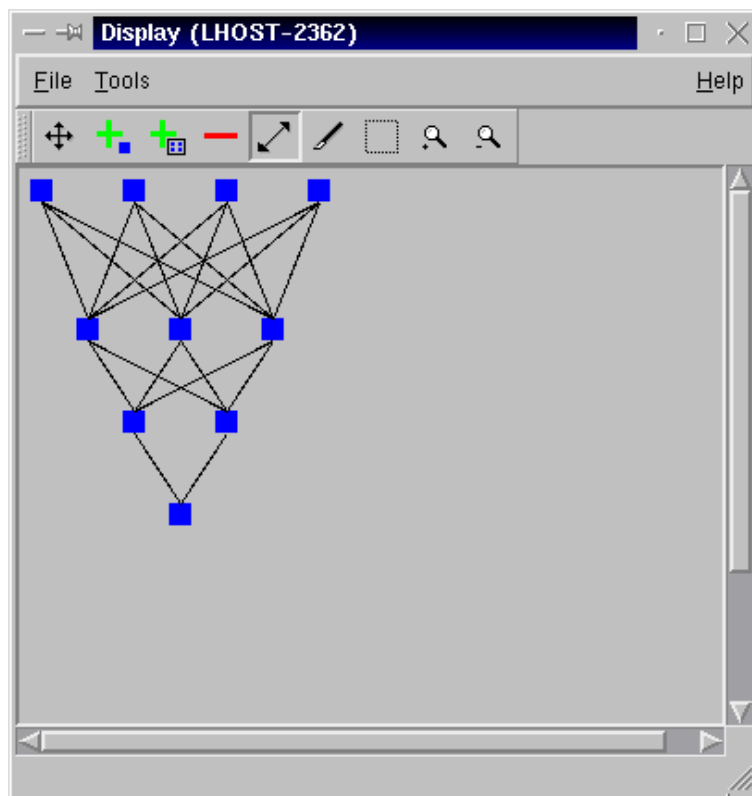


Abbildung 15.2: Das Display in der momentanen 2D-Ausprägung

15.4.1 Grobstruktur

Das Display besteht aus drei lose gekoppelten Teilen: dem Fenster (Klasse: MDisplay), der darin eingelagerten Projektionsfläche (M2DWidget) und dem Entity-Stack (MEntity und abgeleitete Klassen).

MDisplay nimmt dabei die Rolle eines einfachen Wrappers an, der lediglich alle weiteren Elemente enthält. Es beinhaltet ein Menü, einen Toolbar und die Projektionsfläche, die in späteren Versionen per Mausklick ausgetauscht werden kann.

Ein Entity ist in diesem Zusammenhang ein darstellbares Element eines neuronalen Netzes. Also eine Unit, eine Connection oder ein komplettes Netzwerk. In späteren Versionen werden eventuell Entities zur Darstellung von Fuzzy-Systemen hinzukommen.

15.4.2 Die Projektionsfläche

Datei: 2dwidget.h

Bisher existiert nur die 2D-Projektionsfläche (M2DWidget), die lediglich die Z-Koordinate der Entities weglässt und diese dann darstellt. Momentan erfolgt diese Darstellung, indem die Entities aufgefordert werden sich auf dieses Widget zu projizieren.

Die Hauptaufgabe der Projektionsfläche liegt darin die Koordinaten einer Mausektion aufzulösen und an die Entities weiterzugeben. Die Entities entscheiden anhand der mitgelieferten Daten (Toolcode², Koordinaten, gedrückte Maustaste) selbst, ob und wie sie reagieren. Ein einzelnes Entity kann alle Ereignisse für sich beanspruchen (wird zum Beispiel beim Verschieben einer Unit genutzt), indem es sich mit einem Signal der Projektionsfläche verbindet.

15.4.3 Die Entities

Datei: entity.h

Entities sind als eigenständige Objekte definiert, die mit anderen Entities verbunden sein können oder weitere Entities enthalten. Die Basisklasse aller Entities ist die abstrakte Klasse MEntity. Morgana unterscheidet momentan zwischen diesen Entities:

Neuronen (SNNS: Units) sind Entities, die miteinander verbunden werden können, eine feste Größe besitzen und verschoben werden können. Außerdem besitzen sie einen Wert (Activation) und eventuell einen Namen.

Connections (SNNS: Links) definieren ihre Position und Größe durch die Positionen von Anfangs- und End-Neuron. Sie können lediglich erzeugt und gelöscht werden, ihre Position errechnen sie selbst. Connections können einen eigenen Wert besitzen (Weight).

Networks sind Entities, die eine Position und eine Größe besitzen. Sie können eine beliebige Anzahl weiterer Entities enthalten.

Layer sind Entities, die mehrere strukturierte Neuronen darstellen. Sie haben eine Position, eine Größe und speichern die Werte der darin enthaltenen Neuronen. (Layer wurden bisher noch nicht implementiert.)

²Die einzelnen Buttons des Toolbars sind mit numerischen Codes versehen, die die damit verbundene Aufgabe identifizieren

Identifikation

Datei: `netids.h`

Entities werden durch numerische Werte identifiziert. Bei SNNS sind dies einfache int-Werte, die unabhängig voneinander für Units, Layer und Networks eingesetzt werden. Morgana bringt diese Werte in eine Korrelation. Theoretisch können die Units in Morgana in jedem Netzwerk unabhängig von anderen Netzwerken numeriert werden, was SNNS jedoch nicht erlaubt. Da Morgana diese ID's aber nur zur Identifikation nutzt, muss sich der Server um korrekte Nummern kümmern, Morgana selbst macht keinerlei Annahmen dazu³.

Position und Grösse

Datei: `dimtypes.h`

Qt bietet lediglich Klassen mit, denen diskrete zweidimensionale Punkte (Pixel) bezeichnet werden können, Neuronale Netze (zumindest im Model von SNNS und Morgana) haben jedoch dreidimensionale Strukturen und im Display muss teilweise mit Gleitkommawerten gerechnet werden. Daher wurden neue Klassen geschaffen, die Dreidimensionalität und Gleitkomma-Koordinaten abbilden können. Diese Klassen sind untereinander und zu den Qt-Klassen hin konvertierbar und ermöglichen es so Koordinaten sowohl im abstrakten Modell, als auch auf dem Bildschirm zu errechnen und untereinander zu konvertieren.

Bisher sind lediglich die Klassen implementiert, die für eine zweidimensionale Darstellung notwendig sind. Weitere Klassen werden folgen, wenn die dreidimensionale Darstellung implementiert wird.

Nachrichten

Datei: `netevents.h`

Für die Übermittlung von Ereignissen aus der Projektionsfläche an die Entities wird die Klasse `MDrawEvent` genutzt. Sie kann alle vom User ausgelösten und von der Projektionsfläche behandelten Ereignisse speichern. Die Entities müssen anhand der gespeicherten Daten entscheiden, ob sie für das jeweilige Ereignis zuständig sind und wie es zu behandeln ist.

Zur Übermittlung von Nachrichten zwischen Entities und SNNS-Server sind von der abstrakten Klasse `MBaseEvent` abgeleitete Klassen gedacht. Sie können in einem Objekt der Klasse `MEventList` zusammengefasst werden, um möglichst viele Ereignisse auf einmal durch den Entity-Stack zu schleusen. Bisher sind lediglich diese Eventklassen implementiert, für einen realen Datenaustausch und einen Test dieses Entwurfs blieb leider keine Zeit mehr.

15.5 Online-Hilfe

Es wurde versucht ein Hilfesystem zu schaffen, das sowohl für den Nutzer intuitiv, als auch für den Programmierer einfach zu handhaben ist. Außerdem sollte dieses Hilfesystem plattformübergreifend verfügbar sein.

Qt bietet bereits Klassen an, die ein solches System sehr einfach gestalten. So besitzt Qt bereits eine Klasse zur Anzeige von strukturiertem Text sowie Klassen zur Dekodierung von HTML. Da HTML auch sehr einfach in beliebigen Browsern angezeigt, per ASCII-Editor bearbeitet werden kann und bereits den meisten Open Source-Entwicklern bekannt sein dürfte, liegt es nahe diese Funktionalität zu nutzen. Alles was nun noch implementiert werden muss, ist ein Fenster um das Text-Widget herum mit einigen grundlegenden Buttons und Routinen, die die richtigen Dateien zu den jeweiligen Themen zuordnen.

³Ausnahme ist die Reaktion auf Nachrichten vom SNNS-Server, die die Reihenfolge der ID's ändern.

Die exakte Umsetzung der Online-Hilfe ist auf der CD in `/dipl/morgana/gui/helpwin.cpp` zu finden.

Als Programmierer braucht man nichts anderes, als die statische Methode `MHelpWin::contextHelp(QString)` anzuwenden, wobei der Parameter der Dateiname der jeweiligen Hilfedatei relativ zum Hauptverzeichnis der Morgana-Hilfe ist. Die Index-Seite wird zum Beispiel mit `MHelpWin::contextHelp("index.html");` aufgerufen. Beim Schreiben der Hilfetexte sollte man jedoch beachten, dass der Titeltext bereits in der ersten Zeile der HTML-Datei erscheinen muss, damit `MHelpWin` ihn erkennen kann, dieses Problem wird sicherlich in späteren Versionen verschwinden, wenn ein verbesserter HTML-Parser zur Verfügung steht.

Die Implementation des Hilfemenüs wird ebenfalls vereinfacht. Wenn man die Headerdatei `helpmenu.h` eingebunden hat, kann man sich mit der Funktion `helpMenu(QWidget *w, const QString &entry=0, QObject *obj=0, const char *member=0);` ein komplettes Hilfemenü erzeugen. Der Parameter `w` zeigt dabei auf das Fenster, in dem das Menü vorkommen soll, `entry` ist ein Eintrag für dieses Fenster und `obj/member` ein Slot, der reagieren soll, wenn dieser Eintrag ausgewählt wird. Mit dem Makro `HELPMETHODS(classname)` kann man sich automatisch die Slots `HelpIndex`, `helpManual` und `aboutWin` implementieren lassen (die Deklaration lässt sich leider nicht auf diese Weise vereinfachen, da MOC keine Makros auflösen kann).

Intern sind alle offenen Hilfefenster über eine doppelt-verkettete ringförmige Liste verknüpft, damit kann die Methode `contextHelp` immer ein offenes Fenster finden (soweit mindestens eines offen ist), in dem die Darstellung der angeforderten Hilfe erfolgen kann.

Beim ersten Aufruf der Hilfe wird die Methode `findDir` aufgerufen, die in allen Verzeichnissen, die der Wahrscheinlichkeit nach die Hilfedateien enthalten können, nach der Datei `.mdoc` sucht. Das Verzeichnis, das diese Datei enthielt wird in eine statische Variable eingetragen und ab dem Zeitpunkt automatisch genutzt, wenn die Hilfe aufgerufen wird. Ist die Umgebungsvariable `MDOC` gesetzt, wird auch das Verzeichnis mit einbezogen, auf das diese Variable zeigt.

15.6 Probleme

Da die GUI am weitesten fortgeschritten ist haben sich von hier ausgehend auch die meisten Probleme im Design und den vorhandenen Tools gezeigt. Ein großer Teil davon wurde schon in den Abschnitten 13.2 und 14.4 aufgezeigt.

Das `ControlCenter` erscheint trotz aller Verbesserungen noch immer sehr komplex und nur schwierig zu handhaben zu sein. Es ist daher zu überlegen, ob diese Aufteilung der Funktionalität des `ControlCenters` noch optimiert werden kann und ob nicht eventuell einige der Elemente in andere Dialoge oder einfach in die Menüs verschoben werden sollten. Dies lässt sich allerdings erst durch Interaktion mit Nutzern herausfinden, die völlig unvoreingenommen an diesen Dialog herangehen können. Was im momentanen Zustand des Projektes aber kaum möglich sein dürfte, da noch einiges an notwendiger Funktionalität zur Nutzung noch fehlt.

Die Formatierung der Nachrichten in Pakete hat sich auch auf der Client-Seite als relativ problematisch herausgestellt. Es ist nur schwer möglich diese Pakete korrekt in Methodenaufrufe umzusetzen. Ergebnis sind aufwändige Suchläufe, bei denen das zuständige Objekt oft erst ermittelt werden muss (speziell im Fall der Entities, die ja den größten Teil der Kommunikation auf sich ziehen werden).

Die momentane Unterordnung der Entities unter die Projektionsfläche ergibt einige zusätzliche Probleme. Zum einen wird mit jeder Projektionsfläche auch ein kompletter Entity-Stack, also eine teilweise Kopie der Serverdaten, erzeugt. Zum anderen widerspricht die Darstellung der Entities, durch Zugriff der Entities auf die Projektionsfläche, dem Paradigma einer sauberen Trennung von Klassen (Stichwort: Encapsulation).

15.7 Fehlende Elemente

Es fehlen noch sehr viele, teilweise sehr aufwändige, Schritte in Richtung einer funktionierenden GUI. Dies sind einige der wichtigsten:

- Internationalisierung ist zwar bereits vorbereitet, aber bisher nicht umgesetzt. Dies betrifft nicht nur die Einbindung von GNU `gettext`, sondern auch eine länderspezifische Hilfe (die Suchfunktion müsste die Sprache mit einbeziehen) sowie die korrekte Darstellung von Zahlen.
- Die Sessions müssen kontrollierbar werden. D.h. alle notwendigen Parameter (Zugangsberechtigung, Kommunikationskanäle usw.) müssen über Dialoge einstellbar sein.
- Das Display muss noch an den Server angebunden werden.
- Die in 10.4 vorgestellten Diagramme fehlen noch völlig.
- Die für "BigNet" notwendigen Dialoge müssten noch erstellt werden. Es wäre hier auch zu überlegen, ob sie in das Display oder das ControlCenter eingebunden werden sollten. Eventuell sind beide Versionen nötig.

Teil V
Weitere Entwicklungen

Nicht zuletzt dadurch, dass es sich bei Morgana um ein Open Source-Projekt handelt, wird die Entwicklung dieses Programmes weitergehen.

In diesem Teil der Arbeit werde ich Pläne, Ideen und konkrete Designansätze für die weitere Entwicklung darlegen. Das letzte Kapitel dieses Teiles (19 — Projektorganisation) stellt dar, wie das Projekt in Zukunft organisiert werden wird.

Die meisten der hier aufgeführten Vorschläge sind Ansätze, um die im Laufe der Implementierung aufgetretenen Probleme zu lösen. Andere sind gänzlich neue Ansätze, die weitere Entwicklungen anstossen sollen.

Kapitel 16

Inter-Process-Communication

Momentan steht das Morgana-eigene IPC noch am Anfang seiner Entwicklung. Die Verbindungen terminieren teilweise nicht richtig, geringe Datenfehler können die Verbindung völlig durcheinander bringen und es sind keinerlei Sicherheitsmechanismen eingebaut. Es kann sich also jeder mit einem IPC-Server verbinden und die eigentliche Arbeit lahmlegen, indem er sinnlose Kommandos über den Kanal schickt, die den IPC-Kern in undefinierte Zustände bringen.

IPC ist an sich ein paketorientierter Dienst, da in sich abgeschlossene Nachrichten ausgetauscht werden. Gleichzeitig ist es auch ein verbindungsorientierter Dienst, da eine echte Kommunikation stattfindet. Es gibt zwei alternative Protokolle auf Basis von TCP/IP, die die Übertragung von Daten zulassen. TCP (Transmission Control Protocol) ist ein verbindungsorientierter Datenstrom, der von sich aus schon weitgehend absichert, dass die Daten in der richtigen Reihenfolge und korrekt ankommen. UDP dagegen ist, wie das zugrundeliegende IP, verbindungslos und paketorientiert, wobei die maximale Paketgröße der MTU¹ des benutzten Kanals entspricht. Keines der beiden Protokolle entspricht den Anforderungen von Morgana-IPC. TCP bietet keine Pakete und UDP kann nur begrenzt-große Pakete übermitteln und garantiert nicht, dass sie korrekt, in der richtigen Reihenfolge oder überhaupt ankommen. Bei TCP muss also ein Modus geschaffen werden, durch den der Datenstrom in Pakete unterteilt wird und bei UDP müssten Korrektheit, Reihenfolge und Übertragungssicherheit neu implementiert werden. Da TCP die schwierigeren Features schon implementiert hat und die Unterteilung in Pakete mit wenigen Tricks geschehen kann, wird TCP als Übertragungsprotokoll beibehalten werden.

Das IPC wird das erste Tätigkeitsfeld nach Freigabe der Quelltexte sein. Es ist einer der grundlegenden Teile der Morgana-Infrastruktur.

16.1 Änderungen

Am Übertragungsprotokoll müssen einige Veränderungen vorgenommen werden, die die Übertragung an sich absichern. Die hier angeführten Vorschläge sind Designansätze, die bisher nur eine Diskussionsgrundlage bilden. Ob ein Vorschlag implementiert wird und in welcher Form hängt von den Tests während der Implementation ab.

VPackages: Der Datenstrom wird in virtuelle Pakete fester Größe eingeteilt werden, dies stellt sicher, dass einzelne defekte Bytes nicht den ganzen Datenstrom durcheinander bringen. Jede IPC-Nachricht beinhaltet demnach mindestens ein Paket. Jedes Paket trägt dabei eine eigene Sequenznummer und

¹Maximum Transmission Unit, die maximal übermittelbare Blockgröße eines Mediums, bei Ethernet beispielsweise 1500 Byte.

CRC, stimmt die CRC nicht werden nachfolgende Pakete gespeichert und das defekte Paket neu angefordert. Damit dies nicht Denial-of-Service Attacken ermöglicht wird ein "Speicherfenster" eingeführt, das nur eine begrenzte Paketzahl aufnimmt. Dies ist eine Reimplementierung eines Teils der TCP-Funktionalität auf höherer Ebene allerdings in einer sehr begrenzten Version, die lediglich eine zusätzliche Absicherung zu den Absicherungen des TCP/IP-Stacks darstellt und daher nur sehr selten gebraucht werden dürfte. Der zusätzliche Overhead beschränkt sich daher auf wenige Byte pro Paket plus Kontrollstrukturen bei aufgetretenen Fehlern. Die ideale Paketgröße muss noch durch Tests ermittelt werden.

"Kanalbündelung": Es könnten mehrere TCP-Verbindungen pro IPC-Verbindung benutzt werden. Anwendung wäre zum Beispiel die Unterteilung in Kontroll- und Datenkanal. Damit ist sichergestellt, dass nach einem Fehler auf dem Datenkanal immer noch ein virtueller Reset über den Kontrollkanal gesendet werden kann. Es muss dann allerdings ein Optimum zwischen Anzahl der Kanäle und Aufwand zur Verwaltung dieser Kanäle gefunden werden, was auch durch einige Systembeschränkungen beeinflusst werden dürfte.

Redundante Steuerinformation: Alle Steuerinformationen, wie Paketart und Paketgröße sollten redundant übertragen werden, stimmen die zwei oder mehr Versionen nicht überein können diese Informationen neu angefordert werden.

Welche dieser Features auch immer implementiert werden: die Erfahrung zeigt, dass heutige TCP/IP-Implementationen in der Regel relativ stabil und sicher sind, die Dimensionierung dieser Maßnahmen sollte also soweit beschränkt werden, dass sie noch als sinnvoll bezeichnet werden können und nicht zum Selbstzweck werden.

16.2 Klassische Features

Endianess-detection ist ein neues Feature, das hauptsächlich die Geschwindigkeit der Kommunikation erhöhen soll. Bisher werden alle Daten beim Schreiben in ein Paket in Big-Endian konvertiert. Es ist denkbar die Daten direkt in die Pakete zu schreiben und erst auf der Empfängerseite zu entscheiden, ob eine Konvertierung notwendig ist. Dazu wird ein zusätzlicher Schritt im Handshake eingeführt, bei dem standardisierte Werte übertragen und auf der Gegenseite ausgewertet werden, jede Verbindung bekommt intern dann ein Attribut "Endianess", das aussagt, ob eine Konvertierung notwendig ist. Da die meisten genutzten Netze homogen sein dürften, wird hier bei Little-Endian Maschinen die doppelte Konvertierung vermieden.

Message-Queues entsprechen den bisherigen Paketen. Gut wäre jedoch eine zusätzliche Typisierung dieser Pakete, damit die gelieferten Daten nicht fehlinterpretiert werden, wenn unterschiedliche Programmversionen an der Kommunikation beteiligt sind. Dazu müsste ein Modus gefunden werden, mit dem die Namen und Typen der einzelnen Pakete übertragen werden.

Semaphoren, Locks: Oft ist es sinnvoll, dass nur ein Client gleichzeitig auf bestimmte Daten zugreifen kann (z.B. bei sehr großen Strukturänderungen, die aber konsistent bleiben müssen, oder während wichtige Daten in einem Dialog bearbeitet werden). Zur Synchronisierung solcher Ereignisse haben sich auf einzelnen Workstations Semaphoren durchgesetzt, es sollte eine Möglichkeit gefunden werden netzwerk-transparente Semaphoren anzubieten. Die einfachste Lösung wäre den Lock/Semaphor dort zu verwalten, wo er erzeugt wird. Was bei Morgana der Server wäre, da er der einzig sinnvolle Ort für so eine Aktion wäre.

Shared Memory wäre wahrscheinlich nur sehr schwer netzwerk-transparent zu gestalten und zwischen Server und GUI auch kaum sinnvoll, da die gespeicherten Daten hier nicht identisch interpretiert würden. Ob eine Nutzung zwischen zwei Servern in einem Cluster sinnvoll ist, müsste sich erst noch zeigen. Die Synchronisierung der Zugriffe müsste jedenfalls über Semaphoren erfolgen. Der Datenabgleich könnte über Algorithmen wie "rsync" erfolgen, bei denen nur die Teile der Daten übermittelt werden, die sich wirklich geändert haben. Die Implementation eines transparenten Shared Memory wäre also durchaus möglich, aber der Nutzen wäre im Vergleich zum Aufwand sehr zweifelhaft.

Streams, also Datenströme über die große Mengen unformatierter Daten geschickt werden können, könnten zum Beispiel genutzt werden um große Mengen statistischer Daten zwischen Server und GUI zu transportieren. Sie könnten sowohl als zusätzliche TCP/IP-Verbindungen, als auch als virtuelle Verbindungen implementiert werden.

16.3 Objektorientierung

Wie schon in Kapitel 11 beschrieben bieten modernere IPC-Formen die Möglichkeit Objekte über Netzwerke hinweg anzusprechen. Meist ist dazu eine zusätzliche Sprache zur Definition der Interfaces notwendig (z.B. CORBA-IDL). Dazu muss ein zusätzlicher Compiler geschrieben werden, der diese Sprache einliest und Code in der eigentlichen Zielsprache ausgibt (z.B. C++). Qt bietet jedoch ein zusätzliches Feature, das die zusätzliche Sprache weitestgehend unnötig macht: Signale und Slots.

Theoretisch sollte es möglich sein Signale und Slots in zwei getrennten Prozessen (oder gar Rechnern) zu verbinden. Dazu muss eine Kommunikationsschicht geschrieben werden, die die Daten der Signale serialisiert, über die Verbindung schickt und kurz vor dem Aufruf des Slots wieder auspackt. Wenn man den Compiler sparen will, ist dies jedoch mit einer Einschränkung verbunden: die Signale und Slots dürfen nur bestimmte Parameter nutzen. Der hier gezeigte Entwurf versucht diese Einschränkung so weit wie möglich aufzuweichen.

Signal-Slot-Verbindungen können noch recht einfach abgebildet werden. Alles was nötig ist, sind eine zusätzliche connect- und disconnect-Methode im IPC-Layer. Einige der notwendigen Prototypen könnten so aussehen:

```
IPC::connect( IConnection*connection,QString objmethod,
              QObject*local,char*method,
              QString type);
IPC::connect( QObject*local,char*method,
              IConnection*connection,QString objmethod,
              QString type);
IPC::disconnect( IConnection*connection,QString objmethod,
                 QObject*local,char*method,
                 QString type);
IPC::disconnect( QObject*local,char*method,
                 IConnection*connection,QString objmethod,
                 QString type);
```

Wobei `connection` eine offene Verbindung und `objmethod` Objekt und Methode hinter dieser Verbindung darstellen. `local` und `method` sind die lokalen Teilnehmer der Verbindung, `type` die virtuellen Parameter.

Für jede dieser Verbindungen wird eine lokale Signal-Slot-Verbindung auf jeder der beiden Seiten innerhalb des IPC-Layers aufgebaut, die in die dem Nutzer verborgene Transportschicht führt (diese Methode

wird schon von der momentanen Implementation genutzt, wenn nur bestimmte Pakete an ein Objekt geliefert werden sollen). Vor Aufbau der eigentlichen Verbindung müssen Client und Server nur noch überprüfen, ob die Parameter der Methoden stimmen.

Die Typisierung der Verbindungen kann über einen kleinen Trick gestaltet werden. Es wird eine Klasse `IParameter` geschrieben, die sich wie die momentan eingesetzten `IPackage`'s verhält, aber Typisierung unterstützt. Dabei wird einfach neben dem Linksshift-Operator (`<<`) auch der Komma-Operator auf die selbe Art und Weise überschrieben, wenn dann auch noch ein passendes Makro vorhanden ist, würde ein Aufruf in etwa so aussehen:

```
int x=6;
QString s="i'm a string";
emit object.mysignal(IPARAM("VType",x,s));
```

was sich kaum von einem normalen Signalaufruf unterscheidet. Einzig der virtuelle Typ "VType" muss noch deklariert werden (normalerweise unmittelbar vor dem Aufbau einer Verbindung):

```
IPC::registerType(VTYPE("VType","int","QString"));
IPC::connect(&myobject,SIGNAL(mysignal(IParam)),
            connection,":rootobj:network1:neuron3:myslot",
            "VType");
```

wobei `VTYPE` wieder ein Makro ist, das einen String oder ein spezielles Objekt erzeugt, das vom IPC-Layer ausgewertet werden kann.

16.4 Security

Dies sind Vorschläge, die nur in wenigen Fällen überhaupt relevant sein werden. Daher stellen sie nur eine Vervollständigung des Spektrums der Verbesserungen dar.

Multistep Handshake: Momentan benutzt das Morgana-IPC einen Handshake, der noch nicht sehr weit ausgereift ist. Momentan geht der Handshake durch exakt einen Schritt: Absicherung des Protokolls, was bedeutet, dass zwei Strings verglichen werden, einmal die ID des IPC und einmal die ID des nutzenden Programmes. Alles weitere wird bereits im IPC-Modus durchgeführt. Der Handshake sollte wie bei anderen Protokollen auch (z.B. FTP) durch mehrere Schritte gehen: neben dem Abgleich des Protokolls und der Anwendung sollte im Handshake auch ein Abgleich der Protokollfeatures (Schritt 2) und eine Authentifizierung des Clients/Servers stattfinden (Schritt 3).

Encrypted Transfer: gerade bei Kommunikation über große Entfernungen ist die Gefahr hoch, dass die Kommunikation irgendwo belauscht oder manipuliert wird. Es erscheint daher sinnvoll zumindest empfindliche Daten zu verschlüsseln (Authentifizierung, Administrationsdaten). Aber auch hier gilt: es sollte nicht übertrieben werden:

- In den meisten Fällen ist eine Verschlüsselung völlig überflüssig (z.B. bei lokaler Kommunikation oder innerhalb eines überschaubaren Netzwerkes, dem man vertrauen kann).
- Über größere Entfernungen ist meist nur eine Verschlüsselung der sensitiven Daten (Authentifizierung) notwendig, nur wenige verteilte Projekte werden auch alle anderen Daten schützen wollen.

- Bei Netzwerkdaten muss unter Umständen die Herkunft der Daten bestätigt werden, in diesen Fällen reicht bereits ein Hashwert bzw. eine Signatur.

Es sollte also (wenn überhaupt) eine skalierbare Lösung geschaffen werden, die die Nutzer ihren Bedürfnissen anpassen können. Außerdem muss beachtet werden, dass in einigen Ländern die Anwendung von Kryptographie immer noch illegal ist, diese Funktionalität muss also auch komplett abschaltbar sein. Denkbar wäre hier die Nutzung externer Programme, die nur bei Bedarf in den Datenstrom geschaltet werden. Hinzu kommt, dass Kryptographie überdurchschnittlich viel Prozessorlast bedeutet, die Anwendung dieser Verfahren bedeutet also eine nicht unerhebliche Zusatzbelastung für die beteiligten Rechner.

Kapitel 17

Server

17.1 Morgana und SNNS

Der SNNS-Server wird auch weiterhin ausgebaut werden. SNNS wird also weiterhin als der Simulatorkern für Morgana benutzt werden, der die neuesten neuronalen Netzwerkmodelle und neuronalen Algorithmen unterstützt. Allerdings nicht mehr als primärer Server und mit einer massiven Abstraktionsschicht um den SNNS-Kernel herum. Diese Abstraktionsschicht wird all die Features implementieren, die Morgana bietet bzw. benötigt, die aber in SNNS nicht vorhanden sind.

Statische ID's: Das Design von Morgana funktioniert am besten, wenn Units statische, also unveränderliche, ID's besitzen. Der SNNS-Server wird also eine zusätzliche Tabelle anlegen, in der die SNNS-internen ID's auf statische ID's umgesetzt werden. Dazu sollte ein einfaches Array von `short`-Werten genügen.

Hierarchien: Morgana setzt eine hierarchische Beziehung zwischen Subnetz-Nummern und Unit-ID's voraus, der SNNS-Server muss also sicherstellen, dass diese Hierarchien stabil bleiben (SNNS benutzt Subnetz- und Layer-Nummern nur als zusätzliche Parameter in einigen Lernfunktionen, sie können also frei belegt werden).

Objektorientierung: Durch Auswertung der SNNS-internen und der eigenen Tabellen sollte es dem Server möglich sein virtuelle Objekte zu erzeugen, die das Morgana-Modell in SNNS-Funktionsaufrufe umsetzen.

17.2 Ein eigener Simulator

Es ist ein Morgana-eigener Simulatorkern geplant. Er soll direkt das Objektmodell von Morgana umsetzen. Ziel ist es dabei einen Simulator zu schaffen, der durch konsequente Ausnutzung objektorientierter Eigenschaften noch effektiver mit den Systemressourcen umgeht, als SNNS es jetzt schon tut. Außerdem sollen weitere Teilgebiete der künstlichen Intelligenz in diesen Simulator eingebunden werden (soweit dies sinnvoll und technisch möglich ist).

17.2.1 Connection-Machine

Durch die hohe Abstraktion des Morgana-eigenen Modells ist es, zumindest in der Theorie, möglich beliebige Algorithmen als Netzwerk darzustellen, solange sie sich aus Sicht des Simulator-Kerns wie ein Netzwerk verhalten. Zum Beispiel könnten Fuzzy-Systeme als "Netzwerk-Klasse" implementiert werden. Aus Sicht des Simulators werden Daten an Schnittstellen geliefert, die intern nun keine Neuronen, sondern Inputs für das Fuzzy-System sind, und es werden Daten aus dem "Netz" herausgegeben, die den Outputs des Fuzzy-Systems entsprechen anstatt von Neuronen zu stammen.

Ähnliches lässt sich auch mit deterministischen Algorithmen verwirklichen, die sich als Input-Output-Modell darstellen lassen.

17.2.2 Number-Cruncher

Der Morgana-Simulator ist stark objektorientiert. Er kann daher auf der Basis einzelner Objekte bzw. Objekthierarchien parallelisiert werden. Die Hoffnung hinter den folgenden Ansätzen ist es einen leicht auf mehrere Prozessoren/Rechner skalierbaren Simulator schreiben zu können.

Ansatz 1: Der Morgana Simulator geht davon aus, dass jedes Subnetz Input- und Output-Schnittstellen besitzt und dass nach jedem Input ein Output anliegen muss. Damit lässt sich hinter einem Subnetz ein kompletter zweiter Simulator "verstecken". Ein steuernder Simulator könnte auf diese Weise einzelne Bereiche des neuronalen Netzes in weitere Simulatoren auslagern, die eventuell auf entfernten Workstations innerhalb eines LANs ausgeführt werden.

Ansatz 2: Durch die objektorientierte Darstellung des Simulators sollte es kaum Probleme bereiten einzelne Subnetze in eigene Threads zu verlagern. Auf Multiprozessormaschinen können die Prozessoren so optimal ausgenutzt werden, indem mehrere Subnetze parallel bearbeitet werden. Bei bestimmten Subnetztypen und Lernalgorithmen ist es auch möglich innerhalb eines Subnetzes zu parallelisieren.

Ansatz 3: Das Konzept aus Ansatz 2 lässt sich teilweise auf Workstation-Cluster erweitern. Ein einzelner Simulator könnte für den Nutzer transparent auf Prozesse verteilt werden, die über einen Cluster verstreut werden. Dabei werden einzelne Subnetze auf unterschiedliche Prozesse verteilt, die jeder für sich einen eigenen Rechner mit genügend Speicher und einer eigenen CPU besitzen und so sehr aufwendige Berechnungen sehr effektiv durchführen können. Voraussetzung dafür ist jedoch, dass nur Subnetze separiert werden, bei denen der Rechenaufwand den Kommunikationsaufwand stark überwiegt, da die Berechnungen sonst sogar langsamer werden, da die einzelnen Prozesse die Ergebnisse anderer Prozesse abwarten müssen.

Kapitel 18

Oberfläche

Als der für den Nutzer sichtbare Teil von Morgana spielt die GUI insbesondere mit Blick auf Ergonomie und Flexibilität eine große Rolle. Die weitere Entwicklung der GUI kann in diese Ziele unterteilt werden:

18.1 Fertigstellung

Von den in Kapitel 10 vorgestellten Dialogen existieren bisher nur Teile des ControlCenters, des Displays und der Online-Hilfe. Diese müssen nun noch eine Verbindung zum Server bekommen und teilweise noch weiterentwickelt werden. Die anderen dort vorgestellten Dialoge (Pattern Editor und Diagramme) müssen noch implementiert werden, bevor Morgana erste Betatests durchlaufen kann.

Während dieser Betatests wird nicht nur die Funktionalität der GUI, sondern es werden auch die ergonomischen Ansätze der einzelnen Dialoge einer gründlichen Überprüfung unterzogen werden müssen, um eine möglichst gute Anpassung an die Bedürfnisse der Nutzer bewerkstelligen zu können.

18.2 Weiterentwicklungen

Neben der Behebung einiger grundsätzlicher Probleme, wie der Zuordnung der Entities (vgl. Abschnitt 15.6), werden auch weitere neue Schnittstellen benötigt, um angenehm mit Morgana arbeiten zu können.

18.2.1 Dateischnittstelle

Morgana ist bisher nicht in der Lage Daten aus einer Datei zu laden oder in eine Datei zu schreiben. Es sind für die Zukunft zwei unterschiedliche Dateiformate angedacht. Zum einen das SNNS-Dateiformat (siehe [SNNS]), um kompatibel zu SNNS zu bleiben. Zum anderen ein eigenes flexibleres Dateiformat.

SNNS unterscheidet mehrere unterschiedliche Dateiformate. Die Netzwerkdateien sind im Wesentlichen als Text lesbare Abzüge der internen Tabellen des SNNS-Kernels mit den Daten der einzelnen Unit-Typen und spezifischen Werte aller Units. Pattern Files enthalten die Eingangs- und Ausgangswerte aller benutzten Muster eines Netzwerkes. Es existieren noch drei weitere Dateiformate: Result Files (Ergebnisse einzelner Simulationsdurchläufe), Config Files (Einstellungen des ControlCenters) und Logfiles. Zumindest die ersten beiden Dateiformate sollten auch von Morgana unterstützt werden.

Morgana wird nur einen einzigen Datentyp kennen, der flexibel genug ist, um alle anfallenden Daten zu speichern. Da Morgana schon von der grundlegenden Architektur her ein hierarchisches Modell benutzt, liegt es nahe die Daten ebenfalls hierarchisch abzuspeichern. Der Auszug unten demonstriert die Idee der Speicherung, aber besitzt bei weitem noch keine ausgearbeitete Syntax (ich benutze hier eine C-ähnliche Syntax):

```
//Definition der neuronalen Daten:
network(net0){ //dieses Netzwerk umfasst alle Daten
  //Definition einzelner Units
  unit(u1){ //Definition von Unit 0.1 (== network 0, unit 1)
    function=sigmoid;
    type=input;
    activation=1.789;
  }
  unit(u2){
    function=sigmoid;
    activation=0.56;
  }
  unit(u3){
    function=sigmoid;
    activation=0.678;
    type=output;
  }
  //Definition eines Sub-Netzes
  network(net1){
    class=SOM;
    width=20;
    height=20;
    neighborhood=5;
    neighborfunc=gauss;
    weights{ (1.5,1.7,1.88, /*...*/ 1.90),
             (/*...*/), //und so weiter
            }
  }
  //Definition von Connections
  connections{ //Definition: (start,ziel,gewicht)
    (u1,u2,0.5);
    (u1,net1[7,5],2.6);
    (u2,net1[7,6],0.11);
    (net1[15,15],u3,1.5);
  }
} //Ende der Netzwerkdaten

//Definition der Muster (Patterns)
patterns{
  //Definition als ((input),(output))
  ((1.0),(2.0));
  ((3.0),(1.5));
  //usw.
}

//weitere Sektionen koennen folgen...
```

Es können also beliebig komplexe Hierarchien von Netzwerken¹ und Units aufgebaut werden (die anderen Sektionen werden eventuell ähnlich flexibel gestaltet werden). Im Gegensatz zu den SNNS Netzwerkdateien können aber ganze Netze unter Angabe weniger Parameter aufgebaut werden (soweit der Netzwerktyp dies zulässt). Ziel ist es hier ein Dateiformat zu finden, das beide Welten abdeckt: die der automatisch von Morgana erstellten Daten und die der vom Nutzer explizit "programmierten" Netze.

18.2.2 BigNet

SNNS benutzt die Familie der BigNet-Dialoge, um große und komplexe Netzwerke mit regelmäßigen Strukturen aufzubauen. Dabei werden die notwendigen Daten in den Dialogen gesammelt und dann in ein Netzwerk aus unabhängigen Units übersetzt. Morgana wird dieses Konzept nicht mehr kennen. Im Gegensatz zu SNNS besitzen bei Morgana Netzwerke grundsätzlich eine Semantik. Das bedeutet ein regelmäßiger Netzwerktyp kann aufgebaut werden, indem ein neues Subnetz erzeugt wird und ihm eine Netzwerkklasse zugewiesen wird. Die Daten der SNNS-BigNet-Dialoge werden bei Morgana in den Konfigurationsdialogen der Subnetze aufgehen.

18.2.3 Session Management

Momentan kann Morgana lediglich einen Server starten, sich mit ihm verbinden und ihn wieder terminieren. Zukünftig soll Morgana in der Lage sein sich mit schon laufenden Servern zu verbinden und sämtliche Parameter dieser Verbindungen manipulieren können.

Einige dieser Parameter sind:

Verbindungsart: die benutzten Kommunikationswege, also ob Sockets oder Shared Memory (bei lokalen Verbindungen) genutzt werden und welche Ports bei Sockets benutzt werden dürfen (falls eine Firewall zwischen den beiden beteiligten Rechnern steht).

Server-Sockets: die Ports und Interfaces auf denen der Server für weitere Clients zur Verfügung steht. Damit wird eingeschränkt, von welchen Rechnern aus eine neue Verbindung aufgebaut werden kann.

Zugriffsrechte: Passworte und Rechte weiterer Clients. Damit wird effektiv eingeschränkt, welche Nutzer sich an der laufenden Simulation beteiligen dürfen und in welcher Art sie dies tun dürfen.

Parallelisierung: die Art des parallelen Rechnens, die diese Simulation benutzen darf/soll. Siehe auch Abschnitt 17.2.2.

Diese Kontrolle soll zum Beispiel die folgenden Szenarios unterstützen:

Arbeitsgruppen können nun gemeinsam an komplexen Simulationen arbeiten. Jeder bekommt seinen eigenen Zugang auf den Server und die Rechte, die er braucht, um seine Arbeit durchführen zu können.

In Praktika könnten sich beispielsweise Studenten an der Simulation eines Professors einklinken, um auf ihrem eigenen Bildschirm verfolgen zu können, welchen Einfluss die Aktionen des Professors auf die Simulation haben. Diese Idee soll und kann nicht die Projektion an eine Leinwand ersetzen, da die Dialoge nicht automatisch mit den Aktionen des Professors mitgehen würden. Aber die Studenten bekommen so die Möglichkeit über die zentrale Projektion hinaus spezifische Teile der Simulation genauer zu betrachten oder eigene Statistiken anzuzeigen, diese Methode stellt also einen Kompromiss

¹Das Beispiel soll wirklich nur einen möglichen Ansatz für die Syntax aufzeigen. Ein SOM-Netz mit einigen wenigen darum gruppierten Units dürfte nur schwer echte Anwendungen finden.

zwischen eigener Simulation und fest ausgewählten Daten dar. Man ist in der Lage eine Simulation zu nutzen, die zentral durchgeführt wird, ohne selbst wissen zu müssen welche spezifischen Parameter genutzt wurden, und kann trotzdem eigene Auswertungen der Simulationsdaten durchführen.

Komplexe Simulationen können einerseits sehr lange dauern, was es oft nötig macht die GUI zwischenzeitlich zu deaktivieren während die Simulation weiterläuft, und sie können andererseits sehr viel mehr Rechenleistung benötigen, als ein einzelner Rechner zu Verfügung stellt. Es kann also oft notwendig sein den Server in einem Status zu hinterlassen, in dem er weiterrechnet, aber nur bestimmte Personen sich erneut mit dem Server verbinden können (nämlich diejenigen, die das Passwort kennen) und es wird oft auch nötig sein die Rechenleistung ganzer Computernetze zu nutzen.

Kapitel 19

Projektorganisation

Morgana ist bereits auf den Servern von SourceForge abgelegt, von wo aus ab September 2000 die Entwicklung des Projektes gelenkt wird.

Ich beschränke mich hier auf die technischen Aspekte, da sich um Morgana noch keine “sozialen” Strukturen, also eine Entwicklergemeinde, gebildet haben. Eine sehr gute Analyse zu den sozialen Aspekten der Open Source-Entwicklung findet sich in [ESR99] und [WWWESR] jeweils in dem Artikel “Home-steading to the Noosphere”.

19.1 SourceForge

SourceForge ist ein Projekt, das Open Source-Autoren ermöglicht ihre Projekte an einer zentralen Stelle des Internet unterzubringen. Den Projekten werden die Kosten eines eigenen Servers und der größte Teil der Administration abgenommen. Die Autoren können sich also ganz auf Ihre Projekte konzentrieren.

SourceForge bietet den Projekten Services, wie eigene Web-Pages, CVS, FTP, Diskussionsforen und eMail-Verteilerlisten. SourceForge ist über die WWW-Adresse <http://www.sourceforge.net> erreichbar.

Dieses Profil hat Morgana auf SourceForge:

Homepage <http://morgana.sourceforge.net>

anonymous CVS PServer: <cvs.morgana.sourceforge.net:/cvsroot/morgana>

anonymous FTP <ftp://morgana.sourceforge.net>

developers Maillist morgana-devel@lists.sourceforge.net
(Subscription via: morgana-devel-request@lists.sourceforge.net)

Project-Leader Konrad Rosenbaum (pandur@users.sourceforge.net)

Weitere Ressourcen werden mit der Zeit hinzukommen, der aktuelle Stand des Projektes kann auf jeden Fall über die Homepage abgerufen werden. Dort und über die Mailliste(n) kann man auch Kontakt zu den Entwicklern aufnehmen.

19.1.1 Nutzer-Zugang

Alle Morgana-Ressourcen im Web und auf dem FTP-Server können mit einem normalen Web-Browser (anonym) abgerufen werden.

Die aktuellsten Quelltexte können über anonymous-CVS abgerufen werden:

```
cvs -d:pserver:anonymous@cvs.morgana.sourceforge.net login
cvs -d:pserver:anonymous@cvs.morgana.sourceforge.net co morgana
```

wenn CVS hier nach einem Passwort fragen sollte, drücken Sie einfach Enter. Danach kann man von dem Verzeichnis `morgana` aus jederzeit die Quelltexte aktualisieren: `cvs update`. Man sollte sich jedoch darüber im klaren sein, dass es sich bei den CVS-Quellen um Quelltexte handelt, die von den Entwicklern gerade bearbeitet werden. Es ist also durchaus normal, dass diese Quelltexte überhaupt nicht kompiliert werden können oder zumindest fehlerhaft sind. Stabile Versionen werden (sobald die Entwicklung weit genug ist) auf dem FTP-Server des Projektes zum Abruf abgelegt werden.

19.1.2 Entwicklerzugang

Wer sich als Entwickler an dem Projekt Morgana beteiligen will, ist natürlich willkommen. Man sollte jedoch einige Vorarbeit erledigen:

1. Man sollte mit den Quelltexten von Morgana vertraut sein. Im Unterverzeichnis `doc` befindet sich einige Dokumentation, die hier weiterhelfen kann.
2. Die Maillisten des Projektes sind das zentrale Kommunikationsmedium der Entwickler, ein neuer Entwickler sollte also lange genug die Diskussionen mitgelesen haben, um zu Wissen wie die momentane Entwicklung verläuft. Da das Projekt international angelegt ist, wird hier die normale Kommunikationssprache unter Open Source-Entwicklern vorausgesetzt: Englisch.
3. Um als Entwickler eingetragen zu werden, benötigt man einen eigenen Login auf SourceForge. Den kann man sich innerhalb weniger Minuten auf `http://www.sourceforge.net` anlegen (mit einem Klick auf "New User via SSL" kommt man in den entsprechenden Dialog).
4. Zum Schluss kontaktiert man einfach den Projekt-Leiter¹ und bittet um Aufnahme in die Liste der Entwickler.

Für Entwickler ändert sich der CVS-Zugang. Um auch schreibenden Zugriff gewähren zu können wird der Zugang auf einen sicheren Kanal per ssh gelegt. Dazu erzeugt man sich einen ssh-Schlüssel² und speichert den Inhalt der Datei `$HOME/.ssh/identity.pub` in seinem Nutzerprofil auf SourceForge³. Danach setzt man noch diese Shell-Variablen (hier am Beispiel der Bash):

```
export SSH_RSH=ssh
export CVSROOT=loginname@cvs.morgana.sourceforge.net
```

mit dem Kommando

```
cvs co morgana
```

können dann die aktuellen Quellen komplett geladen werden. Dieses Kommando muss auch genutzt werden, wenn neue Verzeichnisse hinzugekommen sind.

¹per eMail an `pandur@users.sourceforge.net`, diese eMail sollte auch den Grund dieser Bitte enthalten, also einen Hinweis auf den Bereich des Projektes, den man bearbeiten will.

²`ssh-keygen`, wenn man `ssh2` benutzt `ssh-keygen1`, da SourceForge Version 1 der SSH voraussetzt

³es sollten keine Zeilenumbrüche entstehen, da der Schlüssel sonst ungültig wird

Weitere wichtige CVS-Kommandos sind:

<code>cvs update</code>	Update der Quelltexte im aktuellen Verzeichnis
<code>cvs commit</code>	Übermitteln der eigenen Änderungen (update sollte vorher aufgerufen werden). CVS fordert zu einem Kommentar der Änderungen auf, dieser Kommentar sollte eine kurze Beschreibung der Änderungen enthalten
<code>cvs add <i>datei</i></code>	Fügt <i>datei</i> der Verwaltung durch CVS hinzu. Beim nächsten <code>commit</code> wird die Datei übertragen
<code>cvs remove <i>datei</i></code>	Löscht <i>datei</i> aus der Quelltext-Verwaltung. Die eigentliche Löschung passiert erst beim nächsten <code>commit</code> .

weitere speziellere Kommandos sind ausführlich in der Dokumentation zu CVS beschrieben ([CVS]).

Eine genauere Beschreibung der Dienste von SourceForge findet sich auf <http://www.sourceforge.net> und den Seiten des SourceForge Documentation Projects ([SFDoc]).

Teil VI

Anhänge

Anhang A

CD-Inhalt

Dies ist eine Auflistung der Verzeichnisse auf der CD, die dieser Arbeit beigeheftet ist:

README Wichtige Hinweise zur CD

dipl/ les- und nutzbare Variante der Diplomarbeit

SNNS/ Quelltexte von SNNS Version 4.2

predoc/ enthält $\text{T}_{\text{E}}\text{X}$ -Datei mit dem Thema der Diplomarbeit

build sh-Script, das SNNS und Morgana compiliert

morgana/ Source-Code von Morgana

ipc/ IPC-Source-Code

gui/ GUI-Source-Code

server/ vorbereitetes Verzeichnis für eigenen Server

snns/ SNNS-Server

include/ zentrale Header-Dateien

doc/ Dokumentationsverzeichnisse

classes/ Quelltextdokumentation der Morgana-internen Klassen

ipc/ Auflistung der IPC-Paketformate

online/ Online-Dokumentation von Morgana

user/ geplant: User-Manual

intl/, po/ vorbereitete Verzeichnisse für die Internationalisierung

lib/ temporäres Verzeichnis für Bibliotheken (SNNS, IPC)

text/ Diplomtexte

diplom.ps druckbare Version der Diplomarbeit, make kann diese Datei neu erzeugen

qtsimple, qtmoc Beispielprogramme, sie können mit `make qt` erstellt werden

vortrag.* Folien eines Probenvortrags im Juli 2000

thesen.ps Diplomthesen zu dieser Arbeit, sie können mit `make thesen.ps` neu erstellt werden

esr_osi/ Texte von Eric S. Raymond und der Open Source Initiative (siehe `index.html` in diesem Verzeichnis)

fsf/ Texte der Free Software Foundation (siehe `index.html` in diesem Verzeichnis)

Anhang B

Glossar

Applikation

einzelnes Programm oder eine Menge von Programmen, die eine bestimmte Funktion erfüllen. Im Kontext der objektorientierten Programmierung auch das Objekt, das alle anderen Objekte initialisiert und die Programmausführung kontrolliert.

Client

Programm, das auf die Dienste eines Servers zugreift. Meistens um sie dem Nutzer zugänglich zu machen.

Cluster (Workstation-²)

Verbund von Rechnern, die über sehr schnelle Netzwerkverbindungen gekoppelt sind und sich in aufwändige Rechenaufgaben hineinteilen.

Connection-Machine

allgemeiner Begriff für Algorithmen, die auf der Verbindung sehr vieler einfacher Elemente beruhen.

CRC (Checksumme)

nach einem bestimmten Algorithmus errechneter Wert, der mit einer bestimmten Wahrscheinlichkeit aussagt, ob die zusammengefassten Daten während eines Transports verfälscht wurden.

CVS (Concurrent Versions System)

ein System, das es Programmierern erlaubt unabhängig voneinander an den selben Quelltexten zu arbeiten. Im Gegensatz zu anderen sog. "Version Management"-Systemen arbeitet CVS aber nicht mit Locks sondern überlässt den Nutzern die Auflösung von Konflikten, dieser Ansatz scheint bei Open Source-Entwicklern sehr beliebt zu sein.

Emulation, Emulator

stellt gegenüber Nutzern und Applikationen eine zusätzliche Schnittstelle zur Verfügung, die eigentlich in dieser Umgebung nicht existiert, oder erzeugt sie mit einem anderen Verfahren.

Endianess

bezeichnet die Darstellungform von Integer-Zahlen im Speicher eines Rechners. Big-Endian-Maschinen stellen dabei immer das höchstwertigste Byte (MSB) und Little-Endian-Maschinen das niederwertigste Byte (LSB) zuerst dar. Zur Gruppe der Big-Endian-Maschinen gehören z.B. Rechner mit Sparc-Prozessoren, zu den Little-Endian-Maschinen die Prozessoren der Intel x86'er Familie.

Fuzzy

ein Gebiet der Künstlichen Intelligenz. Hier werden Entscheidungen mit weniger harten Grenzen

abgebildet, als bei einfachen Wenn-Dann Entscheidungen in der klassischen Programmierung. Es werden Übergänge zwischen absolutem "Ja" und absolutem "Nein" zugelassen.

FTP (File Transfer Protocol)

ein Protokoll mit dem Dateien im Internet transportiert werden. Die spezielle Form des anonymous-FTP ermöglicht es jedem Dateien zu laden, die andere für die Öffentlichkeit bereitgestellt haben.

GUI (Graphical User Interface)

allgemeiner Begriff für grafische Schnittstellen zwischen Mensch und Computer.

Grafikprimitive

die einfachsten darstellbaren Elemente einer grafischen Oberfläche. Dazu gehören z.B. Fenster, Linie, Rechtecke, Buchstaben. Windows bietet dem Programmierer auch Buttons, Menüs und andere komplexe Elemente an, die nicht zu den Primitiven gehören, unter X-Window werden diese Elemente von Bibliotheken aus Primitiven aufgebaut.

IPC (InterProcess Communication)

jede Form von interaktivem Datenaustausch zwischen zwei Prozessen.

LAN

"Local Area Network", siehe *Netzwerk, 2*).

Message-Queue

Systemressource, die es Programmen ermöglicht kurze Datenblöcke (Messages) abzulegen und zur Verarbeitung abzuholen.

Monolith (monolithisches Programm)

Programm aus einem Guss. Also Programm, das in nur einem Prozess abläuft.

Multithreading (siehe auch Thread)

Programmiermethode, bei der ein Prozess in mehrere Threads aufgeteilt wird. Meistens um den (die) Prozessor(en) besser ausnutzen zu können.

Netzwerk, 1)

Neurales Netz, logische Verbindung aus *Neuronen*.

Netzwerk, 2)

Computer-Netz, Kommunikationsverbindung zwischen mehreren Computern. In dieser Arbeit sind damit immer Netzwerke gemeint, die auf dem Internet-Protokoll TCP/IP aufbauen.

Neuron

einzelnes Element eines neuronalen Netzwerks. (siehe [Zell] und Kapitel 2.1 SNNS, Seite 6)

Pipe

virtuelle Datenströme zwischen zwei lokalen Prozessen, die in der Regel nur in eine Richtung Daten transportieren (Halbduplex; *Sockets* werden oft auch als Vollduplex-Pipes bezeichnet). Man unterscheidet zwischen "unnamed" und "named" Pipes. "Named Pipes" sind im Dateisystem sichtbar und können von weiteren Prozessen wie normale Dateien geöffnet werden, "unnamed Pipes" dagegen existieren nur zwischen verwandten Prozessen.

Programm

vom Betriebssystem ausführbare/interpretierbare Datei.

Prozess

Instanz eines Programms im Hauptspeicher eines Rechners. Auf Unix, Windows und anderen sog. Multitasking-Betriebssystemen können mehrere Prozesse gleichzeitig im Speicher existieren, die sich die Ressourcen (wie Speicher, CPU-Zeit und Treiber) des Computers teilen. Prozesse werden vom Betriebssystem voneinander getrennt, d.h. sie können sich nicht direkt gegenseitig beeinflussen (außer über "legale" Kommunikationsmethoden, wie Signale).

Semaphore

Struktur in Betriebssystemen, die sicherstellen soll, dass nur ein Prozess gleichzeitig auf eine Ressource zugreift.

Server

Programm, das anderen Programmen (und damit indirekt den Nutzern) eine Funktion zur Verfügung stellt. Diese Funktion kann sehr einfach¹ aber auch sehr komplex (wie bei Morgana oder einem WWW-Server) sein.

Server-Socket

eine spezielle Form der *Socket*, die nur ein Prozess (der Server) offen hat. Sobald sich ein anderer Prozess (Client) damit verbindet, wird dem Server eine normale Socket übergeben, die eine Verbindung zum Client darstellt, und er kann die Server-Socket weiter nutzen um auf weitere Clients zu warten.

Shared Memory

Speicher (RAM), der von mehreren Prozessen geteilt wird. Meist wird dies als sehr effektive und schnelle Möglichkeit der Kommunikation genutzt.

Socket

Möglichkeit der Kommunikation über TCP/IP Netzwerke und zwischen Programmen auf dem selben (Unix-)Rechner. Dabei wird der Zugriff auf das Netzwerk soweit abstrahiert, dass er kaum noch von der Arbeit mit herkömmlichen Dateien zu unterscheiden ist. Kennzeichnend für Sockets ist, dass jeweils zwei Prozesse damit verbunden sind und jeder Prozess die Daten empfängt, die der jeweils andere sendet. (Spezialfall: *Server-Socket*)

Stream

allgemein für Datenstrom. Sowohl über Netzwerke, als auch zu/von peripheren Geräten oder anderen Prozessen.

Thread (auch: Lightweight Processes; siehe auch Multithreading)

(meistens) Subprozess, der sich zusammen mit anderen Threads den selben Speicher, Dateien und andere Systemressourcen teilt, aber unabhängig ausgeführt wird. Threads können grundsätzlich auf alle Ressourcen zugreifen, die sie sich mit anderen Threads des selben Prozesses teilen.

TCP/IP

Protokollfamilie über die sich Rechner im Internet verständigen können. Dabei bildet IP die unterste Schicht, die Datenpakete von einem Rechner zum nächsten weiterleitet. IP gibt keinerlei Garantien dafür, dass Pakete ankommen und die transportierten Daten noch intakt sind. UDP setzt auf IP auf und transportiert Pakete zwischen einem Client und einem Server. TCP erstellt oberhalb von IP einen Datenstrom, der bis zu einer gewissen Wahrscheinlichkeit absichert, dass alle Daten ankommen und nicht verfälscht wurden (siehe CRC).

Unit

siehe *Neuron*.

Widget

ist ein Kunstwort, das aus "Window" und "Gadget" konstruiert wurde. Ein Widget ist ein eigenständiges Element einer GUI, das kann ein Label, ein Button oder ein ganzes Fenster sein. Widgets können also aus weiteren (Sub-)Widgets bestehen oder eine einzige kleine Funktion ausfüllen.

¹der DayTime Server vieler Unixe antwortet auf Anfragen lediglich mit der momentanen Zeit des Rechners

Literaturverzeichnis

- [CPP] Bjarne Stroustrup *Die C++ Programmiersprache*, Addison-Wesley 1997, ISBN: 3-89319-386-3
- [CVS] Per Cederqvist et al. *CVS-Dokumentation und Info-Pages*
- [ESR99] Eric S. Raymond: *The Cathedral & the Bazaar*, O'Reilly 1999, ISBN: 1-56592-724-9
- [KDE] The K Desktop Environment: <http://www.kde.org>
- [MGTT] U.Drepper, J.Meyering, F.Pinard: *GNU gettext-Manual und Info-Pages*
- [MAC] D.MacKenzie, B.Elliston: *GNU Autoconf Manual und Info-Pages*
- [NET1] W. R. Stevens *UNIX Network Programming — Volume 1*, Prentice-Hall 1998, ISBN 0-13-490012-X
- [QT] Matthias Kalle Dalheimer: *Programming with Qt*, O'Reilly 1999, ISBN: 1-56592-588-2
- [SFDoc] The SourceForge Documentation Project <http://sfdoc.sourceforge.net>
- [ShRun] Robert N. Charrette: *ShadowRun* (Roman-Trilogie; die Bände 2-4, Heyne Verlag, Bestellnummern 06/4845, 06/4846, 06/4847)
- [SNNS] *SNNS – User Manual*
- [SNNS-I] *SNNS – Implementation Manual*
- [TAO-HCI] Brenda Laurel (Editor), et al. *The Art of Human-Computer Interface Design*, Addison-Wesley 1994, ISBN 0-201-51797-3
- [UNIX] W. R. Stevens *Programmierung in der Unix Umgebung*, Addison-Wesley 1995, ISBN 3-89319-814-8
- [UDVR] Unterlagen aus dem Fach “DV-Recht” (Prof. Koitz, HTW-Dresden)
- [UNN] Unterlagen aus dem Fach “Neuronale Netze” Teil I und II (Prof. H. Iwe, HTW-Dresden)
- [USysA] Unterlagen aus dem Fach “System-Analyse” (Prof. J.A. Müller, HTW-Dresden)
- [WWWOS] The Open Source Homepage: <http://www.opensource.org>
- [WWWESR] Writings of Eric S. Raymond: <http://www.tuxedo.org/~esr/writings>
einige der Artikel sind auf der CD unter /esr_osi gespeichert.
- [WWWFSF] The Free Software Foundation: <http://www.gnu.org>
einige der Texte sind auch auf der CD unter /fsf gespeichert.
- [Zell] A. Zell: *Simulation neuronaler Netze*, Oldenbourg 1997, ISBN: 3-486-24350-0

Index

- 2D-Display 37, **55**
- 3D-Display 37

- Aktivierungsfunktion 6
- Applikation 34
- Architektur
 - GUI 54
- Auswertungsphase 7
- Autoconf 27

- batchman 7, 8

- Client 34
- Client/Server 34
- Cluster 70
- COM 41
- Connection 6, 54, 57
- ControlCenter 36, 55
- CORBA 41, 66

- DCOP 41
- Diagramme 39
- Display 37, **55**

- Einfachheit 19
- Endianess 65
- Ergonomie 11
- Erweiterbarkeit 11, 19

- Fehlertoleranz 11

- gettext 26
- Gewicht 6
- GIMP 36
- GNU
 - GNU Autoconf 27
 - GNU gettext 26
- GTK 20
 - GTK— 20
- GUI 54

- Hauptfenster 54
- HTML 40

- Internationalisierung 25
- InterProcessCommunication *siehe* IPC

- IPC 41–42, 48–49, 54, 64–68

- KDE 20, 26
- Kommunikation 41–43, 48, 50
- Kompatibilität 19
- Konsistenz, logische 11

- Layer 57
- Lernverfahren 6
- Link 6, 7, 57
- Lock 65
- Lokalisierung 25

- Main Window *siehe* Hauptfenster
- Message-Queue 41, 65
- Monolith 34
- MPI 41
- MSession, Klasse 54
- Multithreading 34

- Nachricht 41, 42, 48, 58
- Network 57
- Netzwerk 57
- Neuron 6, 57
- neuronale Netze 6, 57

- Oberfläche, grafische *siehe* GUI
- Open Source 12, 63

- Paket 42, 48, 50, 51, 59, 64
- Pattern Editor 37
- Pipe 41
- Portabilität 19
- Prozess 34, 70
- PVM 41

- Qt 20, 40, 42, 48, 66

- Security 67
- Semaphor 41, 65
- Server 34
- Session 35, 54, 73
- Shared-Memory 41, 65
- Site 7
- Socket 41, 48
- SourceForge 75

Stabilität	11
Stream	66
Tcl/Tk	20
Thread	70
Tk	20
Trainingsphase	6
Unit	6, 7, 57
Hidden-	6
Input-	6
Output-	6
Weight	6
xgui	7, 8, 36, 37, 55

Selbständigkeitserklärung

Ich versichere, dass ich die Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Konrad Rosenbaum