Dezentrale Systementwicklung am Beispiel GNU/Linux
Sommersemester 2002

ConfyC

Configure your Configuration Files

Abschlußbericht

1. Oktober 2002

Jan-Ole Beyer, liekedeler@users.sourceforge.net
Thomas Kaschwig, kaschwig@users.sourceforge.net
Matthias Knoll, hanknolo@users.sourceforge.net
Frank Ziegler, fixxz@users.sourceforge.net

Inhaltsverzeichnis

1 Einleitung

Das Projekt ConfyC wurde im Zuge der Veranstaltung *Dezentrale Systementwicklung am Beispiel GNU/Linux* der Technischen Universität Berlin begonnen. Aufgabe der Veranstaltung war es, ein Open-Source-Projekt eigener Wahl zu starten und bis zu einem Prototypen zu führen. Falls es sich herausstellen sollte, dass die Idee nicht zu verwirklichen ist, war als Alternative die Erstellung einer ausführlichen Fehleranalyse möglich. Dieser Fall traf für ConfyC ein. Dieser Abschlußbericht soll also einerseits den Verlauf des Projekts von der Grundidee bis zu den erreichten (und nicht erreichten) Zielen darstellen, andererseits aber auch die gemachten Fehler analysieren und das Projektkonzept in Bezug auf seine Umsetzbarkeit aus heutiger Sicht (nach einem Semester Planungs- und Entwicklungszeit) betrachten.

Zu Beginn soll auf den Ursprung der Idee und auf das Ziel dieses Projektes, das heißt auf die geplanten Leistungen des Programms sowie auf grundlegende Lösungsansätze eingegangen werden. Daraufhin soll der zeitliche Verlauf der Arbeit an ConfyC beschrieben werden.

Nach dieser eher groben Einführung werden dann die theoretischen Grundlagen zu Parser und Interpreter behandelt, das heisst, ausgehend von einer Beschreibung der Struktur von Konfigurationsdateien, kurzen Erläuterungen zu universellen Parsern und Grammatiken insbesondere in Zusammenhang mit diesem Projekt werden sowohl die geplanten Beschreibungsmodule sowie der theoretische Aufbau obiger beider Programmteile erläutert. Im Anschluss daran wird die grafische Benutzerschnittstelle beschrieben — von Vorüberlegungen bis zum Verlauf der Implementierung.

Insbesondere und als wichtiger Teil dieser Arbeit soll dann noch auf die Probleme, die während der Arbeit aufgetaucht sind, eingegangen werden. Sowohl die rein technischen Probleme bei der Implementierung (die primär zum Scheitern führten) als auch Kommunikations- und Koordinationsprobleme werden behandelt. Gerade die Schilderung der technischen Probleme soll auch als Anhaltspunkt dienen, um eventuell nachfolgende Programmierer, die sich für ein solches Projekt interessieren, auf die von uns gewonnenen Erkenntnisse über besondere Schwierigkeiten aufmerksam zu machen, so daß sie nicht „bei Null“ anfangen müssen.

2 Projektziel

Unter UNIX/Linux sind die Konfigurationsdateien im Normalfall einfache Textdateien, die sich per Hand modifizieren lassen — das aber nur mit einer gewissen Vorkenntnis über Aufbau und Struktur der jeweiligen Datei, denn diese sind natürlich keineswegs einheitlich.

Dem Benutzer, der gerade erst anfängt, sich mit diesen Betriebssystemen zu beschäftigen, fällt eine Konfiguration seiner Programme dementsprechend schwer, denn der Umfang und die Möglichkeiten zur Konfiguration sind meist sehr groß. Diese Benutzergruppe kann man unterstützen durch grafische Konfigurationsprogramme. Solche gibt es auch, aber sie sind fast immer Einzellösungen für einzelne Programme, wie zum Beispiel *apacheconf*[1]. Die Entwicklung dieser Programme widerspricht aber im Grunde dem Open-Source-Grundsatz, dass keine Arbeit doppelt gemacht werden sollte. Da sich nämlich die Struktur der Konfigurationsdateien einerseits, aber auch der Aufbau der grafischen Benutzeroberflächen der jeweiligen Programme andererseits sehr ähneln, liegt es nahe, nicht für jedes Programm ein komplettes Konfigurationsprogramm zu schreiben, sondern sich den grafischen „Schnickschnack“ zu sparen und stattdessen auf ein Programm zurückzugreifen, das eine grafische Oberfläche zur Verfügung stellt und sich mit Hilfe verschiedener Module den verschiedenen Dateien anpasst, so dass letztlich so viele Dateien wie möglich bearbeitet werden können.

Diese Aufgabe soll ConfyC, eine Kurzform für *Configure your configuration files*, erfüllen. Verschiedene Module, die jeweils auf eine bestimmte Konfigurationsdatei zugeschnitten sind, wer-

[1] http://www.zecos.com/apacheconf

den es ConfyC erlauben, jedes Programm, für das ein solches Modul existiert, zu konfigurieren. Software-Entwicklern gibt ConfyC die Möglichkeit, einerseits nicht auf eine grafische Konfiguration verzichten zu müssen (was Anfänger von der Software eher fernhalten würde), andererseits aber auch nicht ihre Zeit mit der Entwicklung einer GUI zu „vertrödeln“. Bei Verwendung von ConfyC muss nur ein entsprechendes Modul entwickelt werden — entweder per Hand entsprechend der ConfyC zugrunde liegenden Grammatik oder mit Hilfe von ConfyC selbst, das mit Hilfe eines Standardmoduls erlauben wird, neue Module lediglich „zusammenzuklicken“.

Die ConfyC-Module enthalten aber nicht nur die „interne“ Beschreibung über den Aufbau der Konfigurationsdateien, sondern gleichzeitig eine Beschreibung über Sinn und Zweck der einzelnen Einträge. Damit bietet ConfyC neben der eigentlichen grafischen Konfiguration gleichzeitig eine Online-Hilfe zur Konfiguration, ohne dem Entwickler allzuviel Arbeit zu machen.

Da die Freiheit, Konfigurationsdateien per Hand den letzten Schliff zu geben, so weit wie möglich erhalten bleiben soll, wird einerseits ein Editor in ConfyC eingebunden, anderseits wird darauf geachtet, dass Kommentare und Layout der Dateien nach einer Bearbeitung mit ConfyC erhalten bleiben.

3 Projektentstehung

Die ursprüngliche Projekt-Idee für diesen Kurs war es, ein grafisches Konfigurationstool für den Windowmanager *fvwm2*[2] zu entwickeln. Diese Idee entstand, da zumindest die ursprünglichen drei Gruppenmitglieder Jan-Ole Beyer, Matthias Knoll und Frank Ziegler zu diesem Zeitpunkt den *fvwm2* zwar benutzten und für eigene Zwecke und vor allem Geschmäcker anpassen wollten, aber keine Lust hatten, sich durch die umfangreiche Dokumentation mit all den Möglichkeiten zur Konfiguration, die dieser Windowmanager bietet, zu arbeiten. Sie suchten nach einer einfachen, möglichst grafischen Lösung, die es erlaubt, zumindest eine Grundstruktur zu erstellen, aber deren Details dann auch wieder manuell in der entsprechenden Konfigurationsdatei anzupassen.

Entsprechende Tools für den Vorgänger *fvwm*[3] gibt es — diese sind aber nicht kompatibel zu *fvwm2*. Für letzteren konnte kein Konfigurationstool gefunden werden. Zu diesem Zeitpunkt stieß auch Thomas Kaschwig dazu. Er brachte einige Ideen für andere Projekte mit, die den restlichen Dreien aber weniger interessant (insbesondere im Hinblick auf den Eigennutzen) als das *fvwm2*-Konfigurationstool erschienen. Thomas wollte nicht alleine ein Projekt starten, insbesondere da er es (wie die anderen auch) in einem Kurs über dezentrale Entwicklung für sinnvoller hielt, mit mehreren Leuten an einem Projekt zu arbeiten, und so beschloss er, mit den anderen Dreien zusammen ein Projekt zu beginnen.

Im Zuge der Vorüberlegungen zu einem Konfigurationstool für den *fvwm2*, auf das wir uns nach längeren Diskussionen festgelegt hatten, kamen wir nun zu dem Schluß, dass es wesentlich interessanter und vor allem auch sinnvoller wäre, ein solches Programm nicht nur für diesen einen Windowmanager, sondern auch gleich für seine „Brüder“ *fvwm* und *fvwm95* zu entwickeln. Da sich deren Konfigurationsdateien aber durchaus in vielen Details unterscheiden, nahmen wir das zum Anlass, diese aus diesem Blickpunkt unsinnige Beschränkung aufzugeben und das Projekt auszuweiten auf ein Konfigurationsprogramm für möglichst viele verschiedene Windowmanager. Von dieser Idee war es nicht mehr weit zu ConfyC, denn im Zuge der ersten Überlegungen zu dem Projekt kamen wir zu dem Schluss, dass es kein wesentlich größerer Aufwand sein sollte, die Fähigkeiten des Programms so weit zu erweitern, dass nicht nur Windowmanager, sondern jegliche Programme konfiguriert werden können.

[2] http://www.fvwm.org
[3] http://www.fvwm.org

4 Lösungsansätze

Die ersten konkreten Überlegungen über die Machbarkeit und über Lösungsansätze ließen unzweideutig, dass es sich bei diesem Projekt um ein sehr theorielastiges handeln würde. Der Grundaufbau des Programms sollte wie folgt aussehen: Für jedes Programm, das konfiguriert werden kann, gibt es ein Modul, das sowohl die verschiedenen Konfigurationspunkte als auch die Struktur der zu generierenden Benutzerschnittstelle beinhaltet und so das Arbeiten an der entsprechenden Konfigurationsdatei erlaubt. Diese kann entweder — mit Default-Werten bestückt — neu erstellt oder, wenn sie bereits existiert, eingelesen werden. Der modulare Charakter muss eine besondere Rolle spielen, da auf diese Weise ermöglicht werden soll, immer neue Programme durch einfaches Hinzufügen einer Moduldatei mit ConfyC konfigurieren zu können.

Die Hauptarbeit lag aber eindeutig bei der Erstellung einer Grammatik, die so universell sein muss, dass sie die verschiedensten Konfigurationsdateien zwar beschreiben kann, deren Universalität aber nicht auf Kosten der Details innerhalb der Konfigurationsdateien gehen darf. Auf dieser Grundlage einer solchen Grammatik sollten dann Scanner, Parser und Interpreter entwickelt werden. Außerdem musste eine dynamische GUI erstellt werden, die es erlaubt, dem Benutzer passend zum jeweiligen Modul — also extrem flexibel — eine sinnvolle Anordnung an Buttons, Textfeldern usw. zu bieten, so dass eine möglichst intuitive Arbeit möglich wird.

Eine solche Grammatik sollte sich im Laufe der Weiterentwicklung aber als extrem komplex erweisen. Zwar lassen sich die verschiedensten Konfigurationsdateien „im Großen und Ganzen“ recht gut in eine Grammatik hineinzwängen, aber wie so häufig steckt der Teufel im Detail. Die Probleme, die dadurch entstanden und die Idee eines universellen Konfigurationstools schließlich völlig aus dem Kosten-Nutzen-Gleichgewicht brachten, werden insbesondere im Abschnitt „Technische Probleme“ betrachtet.

Als Entwicklungssprache wurde C++ insbesondere wegen der bei einem solchen Projekt sehr hilfreichen Objektorientiertheit gewählt, zur Entwicklung der GUI sollte auf Qt[4] zurückgegriffen werden — dies ebenso aus dem Grunde der Objektorientiertheit dieses GUI Toolkits.

Auf nähere Details zur GUI, aber auch zur Entwicklung der Grammatik und zum Parser — der letztlich zum Scheitern des Projekts führte — wird in den entsprechenden Abschnitten noch näher eingegangen. Der Interpreter dagegen wird nicht näher betrachtet werden, da das Projekt aufgrund der Probleme bei der Implementierung des Parsers ins Stocken kam und somit der Interpreter (ohne Parser) nicht weiter betrachtet wurde.

5 Projektverlauf

Der Ursprung der Idee zu ConfyC wurde weiter oben schon betrachtet. An dieser Stelle soll nun in einem kurzen Abriss der Verlauf der Projektarbeit beschrieben werden.

Eine Teilaufgabe der Veranstaltung war es, eine Internetpräsenz mit Codeverwaltungssystem, Download-, Kommunikationsmöglichkeiten usw. aufzubauen. Zu diesem Zweck wählten wir als Plattform *Sourceforge*[5] aus — genausogut hätte es auch ein anderer Anbieter wie zum Beispiel *BerliOS*[6] sein können. Wider Erwarten wurde unsere Projektbeschreibung schon nach wenigen Stunden von *Sourceforge* akzeptiert, so dass wir uns bereits recht schnell in die Möglichkeiten, die uns geboten wurden, einarbeiten konnten. Um eine Kommunikation zwischen den Gruppenmitgliedern zu ermöglichen, wurde eine nicht-öffentliche Development-Mailinglist eingerichtet. Öffentliche

[4] http://www.trolltech.com/products/qt
[5] http://www.sourceforge.net
[6] http://www.berlios.de

Mailinglisten sollten folgen, sobald ein erstes Release von ConfyC veröffentlicht wird. Dass es hierzu nicht kam, wurde bereits in der Einleitung erwähnt.

Während Jan-Ole sich primär noch mit *Sourceforge* beschäftigte, um dem Rest der Gruppe eine einfache Einführung in die Arbeitsweise mit CVS etc. geben zu können, beschäftigte sich Frank bereits mit grundlegenden Überlegungen zur Erstellung der notwendigen Grammatik. Diese Arbeit sollte sich anfangs hauptsächlich auf das Auswerten der verschiedensten Konfigurationsdateien beschränken, um überhaupt einen Überblick zu bekommen über die vorhandenen (und zu bewältigenden) Strukturen innerhalb der verschiedenen Dateien. Matthias und Thomas begannen, sich in Qt einzuarbeiten, da beide noch keine Erfahrung damit hatten.

In etwa zeitgleich wurde die Arbeit an der Website begonnen. Sie sollte sich zu Beginn auf die wichtigsten Punkte beschränken. Auf Grafiken, Logos etc. wurde zu Gunsten einer verstärkten Arbeit an den theoretischen Grundlagen verzichtet. Im Laufe der Entwicklungszeit sollte sie aber überarbeitet werden. Dass es dazu nicht kam, liegt am ursprünglich wesentlich geringer geschätzten Arbeitsaufwand beim Versuch, das Projekt trotz aller Probleme weiterzuführen.

Nachdem die theoretischen Überlegungen zu einer ersten Version einer Grammatik geführt hatten, sollte die eigentlich Implementierung beginnen.

Da Frank sich bereits ausgiebig mit den theroretischen Grundlagen, der Grammatik und den verschiedensten Konfigurationsdateien beschäftigt hatte, begann er, Scanner und Parser zu implementieren. Thomas und Matthias sollten sich ursprünglich mit der GUI beschäftigen; da sich aber die Arbeit hier schlecht teilen ließ, arbeitete nur Thomas an der GUI, während Matthias zusammen mit Jan-Ole die Implementierung des Interpreters übernehmen sollte — Matthias eher im Bereich der Schnittstelle zur GUI, Jan-Ole dagegen an der eigentlichen Funktionalität.

Die Arbeit an der GUI kam schnell vorwärts bis zu dem Zeitpunkt, zu dem der Interpreter bzw. die Schnittstellen von diesem benötigt wurden. Der Interpreter jedoch war noch nicht begonnen worden, da Frank schon zu Beginn der Implementierung des Parsers auf Probleme stieß, die sich bis auf weiteres nicht lösen ließen und ohne deren Datenstruktur die Arbeit am Interpreter nicht begonnen werden konnte. Hierauf wird in den entsprechenden Kapiteln noch näher eingegangen.

Da sich bis zu diesem Zeitpunkt aber nur Frank mit der Theorie beschäftigt hatte, konnten die restlichen Gruppenmitglieder weder an ihren Programmteilen weiterarbeiten noch Frank „auf die Schnelle“ helfen, da der theoretische Bereich mittlerweile sehr komplex geworden war. Diese Problematik führte zwangsläufig zu einer Stagnation einerseits der Entwicklung der restlichen Teile, andererseits aber auch der „Verschönerung“ der Website etc. Da es Anfang September schließlich immer unwahrscheinlicher wurde, einen Prototyp noch bis zum ersten Oktober fertigzustellen, beschlossen wir, das Projekt stillzulegen und uns stattdessen mit gemachten Fehlern zu beschäftigen — sowohl mit den technischen beim Bau des Parsers als auch bei kommunikativ-koordinatorischen. Ein erneuter Versuch, ConfyC doch noch fertigzustellen, wird aller Voraussicht nach nicht stattfinden.

6 Theoretische Grundlagen von Parser und Interpreter

6.1 Die Struktur von Konfigurationsdateien

Um eine Grundlage für die Verarbeitung von Konfigurationsdateien zu schaffen, muss zu erst deren Struktur analysiert werden. Auf dieser Grundlage kann dann eine allgemeine Grammatik entwickelt werden, die wiederum die Basis für die Parser- und Interpretermodule bildet.

Es gibt zwei Kategorien von Konfigurationsdateien. Die einen sind sehr simpel und bestehen

nur aus einer Sammlung von Konfigurationszeilen, die untereinander keinen Zusammenhang zeigen und nur Optionsvariablen Werte zuweisen. Die anderen sind hierarchisch aufgebaut, in teilweise verschachtelte Sektionen unterteilt, die wiederum Sammlungen von Konfigurationszeilen enthalten. Nun ist es aber möglich, im ersten Fall die gesamte Datei als eine Sektion zu betrachten, wodurch beide Kategorien zu einer verschmelzen. Diese Betrachtungsweise ist wichtig, da nun davon ausgegangen werden muss, dass es kein explizites Erkennungsmerkmal für Sektionen gibt bzw. geben muss. Natürlich sind in den Dateien, die sich in Sektionen gliedern, diese auch speziell gekennzeichnet. Meist findet dies durch Klammerung, teilweise durch Namensgebung, häufig durch Schlüsselwörter statt. Ein einheitliches Muster ist nicht vorhanden.

Viele Konfigurationsdateien erlauben Kommentare. Diese werden meist durch spezielle Zeichen eingeleitet und enden entweder am Ende der Zeile oder — seltener — durch eine abschließende Zeichenfolge.

Der wichtigste Aspekt im Aufbau der Konfigurationsdateien ist allerdings die Konfigurationszeile selbst. Zwar ist die einfache Verknüpfung von Optionsname und Wert sehr häufig, aber es gibt daneben auch sehr viele Variationen. So können in manchen Dateien Optionen mehrere Werte beinhalten, in anderen muss es wiederum genau ein Wert sein. In einigen Dateien können Werte von Optionen Argumente sein, die sich auch wie Optionen aufbauen oder sogar eigenständige Optionen sind, die auch an anderer Stelle in der Datei unabhängig auftreten könnten.

Es gibt auch eine ganze Reihe von Optionsparametern, also zulässigen Wertetypen. Es kommen numerische Argumente, Zeichenketten oder Zeichenketten in Anführungszeichen vor. Zeichenketten können Namen beinhalten oder aber Schlüsselwörter sein. Auch können sie Bezüge herstellen, zum Beispiel auf Dateien verweisen.

Klammern, Trennzeichen und sonstige Syntax ist von Konfigurationsdatei zu Konfigurationsdatei teilweise grundverschieden. Verknüpfungen zwischen Option und Wert können beispielsweise durch Gleichheitszeichen, Doppelpunkte oder auch ohne jegliche Zeichen erfolgen, um nur einige Varianten zu nennen.

Generell ist festzustellen, dass es eine ungeheure Vielzahl an Unterschieden zwischen den Konfigurationsdateien gibt, die sich aber zumeist auf die syntaktische Seite beschränken. Das Überbrücken dieser Unterschiede stellt einerseits eine der größeren Schwierigkeiten dar, andererseits ist dieses eine der Hauptzielstellungen des Projekts. Dieses Projekt wäre allerdings fast überflüssig und auf jeden Fall leichter realisierbar, wenn es einheitliche Struktur- und Syntaxmerkmale in den Konfigurationsdateien geben würde.

6.2 Universeller Parser

Als universellen Parser bezeichnen wir eine Software, die in der Lage ist, jegliche Dateien in eine einzige, sinnvolle, aber wiederum universelle und allgemein definierte Datenstruktur einzulesen. Diese Software ist nicht nur ein eigentlicher Parser im engeren Sinne, sondern umfasst auch den dazugehörigen Scanner, der eine Datei in einen für den eigentlichen Parser verarbeitbaren Tokenstrom umwandelt. Der Parser muss nicht den Sinn hinter der einzulesenden Dateistruktur erkennen, sondern nur den Aufbau der Dateistruktur selbst, um sie in die definierte allgemeine Datenstruktur einlesen zu können.

Für unser Projekt ist kein wirklich universeller Parser notwendig, da es sich bei den von uns zu verarbeitenden Dateien um eine spezielle und genau definierte Gruppe von Dateien handelt, die Gruppe der Konfigurationsdateien. Daher muss der Parser nur partiell universell sein, also alle Dateien, die dieser Gruppe angehören, verarbeiten können. Insofern kann auch eine spezieller definierte Datenstruktur verwendet werden, die nur insofern allgemein und universell ist, als dass alle Daten, die in dieser Gruppe von Dateien entstehen können von ihr aufgenommen werden können

und von unserer Software verarbeitbar bleiben.

Als weitere Einschränkung zur Universalität soll der Parser des Projektes auf Vorwissen zurückgreifen. Dieses Vorwissen wird von den Modulen geliefert, welche die zu verarbeitenden Dateitypen jeweils näher beschreiben.

Dieses Vorwissen ist vor allem auch für den Scanner notwendig, da sich hier auch die Symbole definieren, welche die Abgrenzungen und Art der Tokens bestimmen, denn diese Symbole können von Konfigurationsdatei zu Konfigurationsdatei unterschiedlich sein.

6.3 Universelle Grammatik

Ein universeller Parser benötigt als Arbeitsgrundlage eine universelle Grammatik. Diese Grammatik beschreibt normalerweise sowohl die Dateistruktur, als auch die Datenstruktur, beziehungsweise wird die Datenstruktur durch die Grammatik bestimmt. Eine wirklich universelle Grammatik müsste alle nur möglichen Dateistrukturen in möglichst sinnvoller Weise erfassen, um den eigentlichen Zweck, die Weiterverarbeitung mittels Interpreter, zu gut wie möglich zu gewährleisten.

Da wir nur einen partiell universellen Parser benötigen, bedingt durch unser begrenztes Problemfeld, brauchen wir auch nur eine partiell universelle Grammatik. Diese soll die Struktur aller nur denkbaren Konfigurationsdateien in einer allgemeingültigen Struktur erfassen. Das heißt zum einen, dass die Struktur der Grammatik so aufgebaut sein soll, dass alle in Konfigurationsdateien vorkommenden Strukturen sinnvoll erfasst werden. Zum zweiten sollten Konfigurationsdateien, denen Strukturen fehlen oder in denen gar keine Strukturierung vorkommt, ebenso erfasst werden, und letztendlich soll der Hauptbestandteil von Konfigurationsdateien, nämlich die Konfigurationselemente, möglichst sinnvoll, aber auch möglichst einfach darzustellen sein. Eine weitere Anforderung an unsere Grammatik ist, dass auch die Konfigurationsmodule für unser Programm genauso sinnvoll darstellbar sind. Dies ist natürlich gleichzeitig eine Anforderung für den Aufbau der Module, die ebenfalls universell Konfigurationsdateien in ihrer Struktur, in ihrem Sinnzusammenhang und ihren Besonderheiten beschreiben sollen, denn die Moduldaten müssen von Scanner, Parser und Interpreter gemeinsam benutzt werden und vom Parser eingelesen werden.

Die Grammatik basiert darauf, dass sie die Dateien zu erst in beliebig viele, beliebig verschachtelte Sektionen aufteilt. Da diese Sektionen sehr unterschiedlich aussehen können, beziehungsweise, wie die meisten Grammatikelemente, nicht einmal real vorhanden sein müssen, setzt sich eine Sektion aus etlichen, teils optionalen Teilkomponenten zusammen. Diese Teilkomponenten werden zunächst in Sektionskopf, Sektionsrumpf und Sektionsfuß zusammengefasst. Sektionskopf und Sektionsfuß werden wiederum aus einer Vielzahl von optionalen Teilen gebildet. Hierbei relativ wichtig sind die lead- und follow-chars. Diese gleichen die syntaktischen Unterschiede zwischen den verschiedenen Konfigurationsdateien aus, sind aber für die Sinnzusammenhänge in der Datenstruktur völlig unbedeutend. Weiterhin können Sektionen mit Schlüsselwörtern eingeleitet und/oder durch Namen identifiziert werden.

Der Sektionsrumpf enthält die an sich wichtigsten Elemente — beliebig viele Konfigurationszeilen. Diese enthalten jeweils eine Option und, ähnlich den lead- und follow-chars im Sektionskopf und -rumpf, auch optionale Zeichenfolgen am Anfang und Ende, um syntaktische Unterschiede abfangen zu können. Die Option ist das kompliziertesteste Objekt, da sie auf sehr verschiedene Art und Weise aufgebaut sein kann und in der Grammatik eine universelle und allgemeine Beschreibung dafür vorhanden sein muss, in der jeweils sinngleiche Elemente immer denselben Grammatikelementen zugeordnet werden. Dabei kann sowohl die Anzahl als auch teilweise die Anordnung der Elemente variieren.

Die folgende Grammatik haben wir als Grundlage für unser Projekt und die beschriebenen Anforderungen entwickelt:

ConfyC - configuration file grammar:

```
<configfile>               = <section>
<section>                  = <section_declaration> <insection>
<section>                  = ε
<insection>                = <section>
<section_declaration>      = <section_header> <section_body> <section_footer>
<section_header>           = <sh_lead_chars> <sh_keyword> <sh_ident> <optional_flags>
                             <sh_follow_chars>
<section_header>           = ε
<section_footer>           = <sf_lead_chars> <optional_footer_statement> <sf_follow_chars>
<section_footer>           = ε
<section_body>             = <configuration_line> <section_body>
<section_body>             = ε
<sh_lead_chars>            = ide
<sh_lead_chars>            = ε
<sh_follow_chars>          = ide
<sh_follow_chars>          = ε
<sf_lead_chars>            = ide
<sf_lead_chars>            = ε
<sf_follow_chars>          = ide
<sf_follow_chars>          = ε
<sh_keyword>               = ide
<sh_ident>                 = ide
<sh_ident>                 = ε
<optional_flags>           = <flag> <optional_flags>
<optional_flags>           = ε
<flag>                     = ide
<optional_footer_statement> = ide
<configuration_line>       = <optional_leading_char> <option> <optional_closing_char>
<optional_leading_char>    = ide
<optional_leading_char>    = ε
<optional_closing_char>    = ide
<optional_closing_char>    = ε
<option>                   = <op_name> <optional_connector> <op_list>
<op_name>                  = ide
<optional_connector>       = ide
<optional_connector>       = ε
<op_list>                  = <op_flag> <op_list_cont>
<op_flag>                  = ide
<op_flag>                  = <option>
<op_flag>                  = number
<op_flag>                  = ε
<op_list_cont>             = <optional_divide> <op_flag> <op_list_cont>
<op_list_cont>             = ε
<optional_divide>          = ide
<optional_divide>          = ε
ide                        = keyword
ide                        = string
```

Die mit „ide“ bezeichneten Identifier sind Schlüsselwörter und identifizierende Zeichenketten, die vor allem für den Scanner Bedeutung haben, weil von ihnen der Aufbau des Tokenstroms abhängt.

Leider ist es uns bisher nicht gelungen, eine Methode zu finden, die sicherstellt, dass die vorgestellte Grammatik die Anforderungen auch wirklich erfüllt. Sie wurde vielmehr durch viel Probieren anhand verschiedenster Konfigurationsdateien erstellt.

6.4 Die Beschreibungsmodule

Neben dem Entwurf der Grammatik beschäftigten wir uns parallel damit, eine sinnvolle Form für die Beschreibungsmodule zu entwickeln. Diese ermöglichen die eigentliche Funktionalität der Software, indem sie den Aufbau als auch die Semantik der Konfigurationsdateien, die mit ConfyC bearbeitet werden sollen, formal beschreiben. Hierbei haben wir zwei vorrangige Ziele verfolgt. Zum einen sollten die Module übersichtlich zu erstellen sein, sich aber auch am Aufbau der Grammatik orientieren, damit das Zusammenspiel zwischen den Modulen und der vom Parser und Interpreter benutzten Datenstruktur leichter fällt. Zum anderen sollte das Modul selbst vollständig und sinnvoll durch unsere Grammatik darstellbar sein, damit der Parser auch für die Module verwendbar ist.

Die Module sollen aber nicht nur vom Parser eingelesen werden, sie müssen auch vom Parser benutzt werden, um Konfigurationsdateien lesen zu können. Sie enthalten nämlich die Syntaxelemente für die spezifischen Dateitypen und definieren Stringformate, Terminalsymbole und vor allem leere Elemente.

Da die Grammatik davon ausgeht, dass die Konfigurationsdateien in Sektionen unterteilt sind, werden in den Beschreibungsmodulen alle Sektionen separat definiert. In anfänglichen Entwürfen fiel schnell auf, dass viele Optionen, die definiert werden mussten, in vielen Sektionen wiederholt auftraten und jedes Mal neu zu beschreiben waren. Daher bildete sich im Laufe der Entwicklungszeit ein Format heraus, in dem erst alle Optionen definiert werden und dann die Sektionen, in denen nur noch erwähnt wird, welche Optionen in der Sektion vorkommen dürfen.

Zusätzlich sollen die Beschreibungsmodule auch semantische Elemente enthalten. Das bedeutet, sie beschreiben neben den Optionen oder Sektionen selbst auch deren Bedeutung. Sie beinhalten Beschreibungen, die später vom Programm auf der graphischen Oberfläche angezeigt werden, um den Benutzer durch die Konfiguration zu führen.

Das erste Format, welches noch komplett sektionsweise definierte, sieht wie folgt aus (Beispielmodul für einen Windowmanager, „#“ kennzeichnet eine Kommentarzeile):

```
<section Style>
        # Beschreibung
        description = "Window specific options"
        # Es gibt keine section_header/section_footer
        section_header = empty
        section_footer = empty
        # "Style" leitet die configuration_line ein
        optional_leading_char = "Style"
        # die configuration_line endet mit einem carriage return
        optional_closing_char = newline
        # Es gibt keine Verschachtelungen in dieser Sektion
        insection = empty
        # Es gibt nur eine configuraton_line in dieser Sektion
        section_body = empty
        # Es folgen die möglichen Optionen
        <option StyleOptions>
                description = "Window options/modifier"
                op_name = string
                optional_connector = empty
                optional_divide = ","
                <op_flag NoHandles>
                        description = "Keine Buttons"
                </op_flag>
                <op_flag Sticky>
                        description = "Bleibt auf Desktop"
                </op_flag>
                <op_flag>
                        .
                        .
                        .
                <op_flag BorderWidth>
                        <option BorderWidth>
                                description = "Dicke des Randes"
                                op_name = "BorderWidth"
                                optional_connector = empty
                                op_list_cont = emtpy
                                op_flag = number
                        </option>
                        description = "Dicke des Fensterrandes"
                </op_flag>
        </option>
        <option WindowIcon>
                        .
                        .
                        .
        </option>
                .
                .
                .
</section>
```

Eine weiterentwickelte Form des Moduls definiert eine zusätzliche Sektion „Common“, die jetzt alle möglichen Optionen enthält. Dadurch entfallen viele der Verschachtelungen. Ansonsten wird die Form fast unverändert beibehalten:

```
<section common>
        sectionheader = empty
        sectionfooter = empty
        <sectionbody>
                <option KillModule>
                        option = FvwmAuto
                        option = FvwmButtons
                               .
                               .
                </option>
                <option Key>
                        optionalconnector = empty
                        # Flag an erster Position
                        <opflag 1>
                                option = F1
                                       .
                                       .
                                option = F10
                        </opflag>
                        # Flag an zweiter Position
                        <opflag 2>
                                option = A
                                option = N
                                       .
                                       .
                        </opflag>
                        </option>
                            .
                            .
        </sectionbody>
        <insection>
                option = Menu
                       .
                       .
        </insection>
</section>
<section Menu>
        insection = empty
        <sectionheader>
               .
               .
        </sectionheader>
               .
               .
</section>
     .
     .
```

Auch wenn es per Hand immer noch eine Menge Arbeit macht, ein solches Modul zu schreiben, so ist es in der letzten Form relativ effizient möglich. Außerdem soll eigentlich nur das ConfyC-Standardmodul von Hand geschrieben werden, alle weiteren Module werden mit der Software und der graphischen Oberfläche erzeugt. Hier ist auch zu überlegen, ob das Standardmodul zu beschreiben ist oder ob es nicht einfach in die Software integriert werden kann.

6.5 Theoretischer Aufbau und Leistungsumfang des Parsers

Der Parser unterscheidet sich von einem durchschnittlichen Parser für eine kontextfreie Grammatik nur in einem Punkt: Es gibt keine festen Terminalsymbole in unserer universellen Grammatik. Dies ist allerdings sehr entscheidend, vor allem für die Umsetzbarkeit in eine konkrete Implementierung. Hierbei ist es notwendig, dass sich sowohl Scanner als auch der eigentliche Parser während des Einlesens der zu verarbeitenden Dateien abhängig von ihrem Zustand anhand der Informationen über die Datei, welche aus dem jeweiligen Modul entnommen werden können, selbst konfigurieren. Dabei müssen nicht nur die Terminalsymbole redefiniert werden, sondern auch die Form der Stringdarstellung und der Eigennamendarstellung sowie die Schlüsselwörter. Da die Sektionen der Konfigurationsdateien jeweils andere syntaktische Eigenschaften aufweisen können, muss in jeder Sektion neu redefiniert werden. Dafür muss zuvor eindeutig erkannt werden, welche Sektion gerade eingelesen wird. Die Informationen, die zur Unterscheidung benötigt werden, stammen aus dem Beschreibungsmodul. Dieses kann nur gelesen werden, wenn es eine Standardkonfiguration für den Parser gibt, mit der die Syntax der Beschreibungsmodule geparst werden kann. Diese Standardkonfiguration kann also direkt in den Parser eingebaut werden. Damit ist es fast schon sinnvoll, das gesamte ConfyC-Standardmodul in die Software direkt zu integrieren.

Es ist sinnvoll den Scanner und den eigentlichen Parser getrennt zu betrachten, da der Scanner noch etwas einfacher realisierbar sein sollte. Der Scanner soll dem Parser einen Tokenstrom liefern. Die Tokens enthalten schon einige spezifische Informationen. Es werden Schlüsselwörter, Strings, eventuell Quoted Strings, Zahlen und alle vorkommenden Arten von Separatoren unterschieden. Hier ist vor allem die Unterscheidung von Schlüsselwörtern und Strings sowie die Einordnung der Separatoren problematisch. Da die Tokens im allgemeinen über alle Sektionen in ihrer Form gleich sind und Schlüsselwörter normalerweise global definiert sind, empfiehlt sich eine separate Sektion im Beschreibungsmodul, die nur für den Scanner benutzt wird, um die genannten Unterscheidungen treffen zu können. Wir haben diese Sektion noch nicht realisiert. Sie kann allerdings nicht im Parser verwendet werden, da hier noch eine weitere Unterscheidung in gültige und nicht gültige Schlüsselwörter erfolgen muss, die von Sektion zu Sektion verschieden ausfallen kann.

Der Parser kann also nicht global eingestellt werden, sondern muss von Sektion zu Sektion entscheiden können, welche gültigen Konfigurationen auftreten können. Das Problem ist hier nicht nur die Implementierung des Parsers, sondern vor allem eine sinnvolle Unterbringung der vom Parser benötigten Informationen in den Beschreibungsmodulen. Es muss sichergestellt werden, dass diese Informationen ausreichen, um alle nötigen Entscheidungen treffen zu können.

Das Parsen einer Datei gliedert sich also in das Parsen der einzelnen Sektionen, die auch in der Datenstruktur vorhanden sein sollen. Um wiederum eine Sektion zu parsen, müssen die zugehörigen Informationen aus dem Beschreibungsmodul eingelesen werden und der Parser für die Sektion konfiguriert werden. Die Daten des Beschreibungsmoduls sollten natürlich auf einmal eingelesen werden und dann die entsprechende Teilstruktur abgerufen werden. Anhand dieser Informationen werden nun gültige und ungültige Fälle unterscheidbar und man erhält einen ganz normalen Rekursivparser, der nach Fallunterscheidungen die jeweiligen Grammatikstrukturen parst, bis die Sektion beendet ist und der Parser neu eingestellt wird oder aber bis eine neue Sektion entdeckt wird, die in der zu parsenden Sektion enthalten ist. Dann muss für diese innere Sektion der Parser neu konfiguriert werden, die Sektion geparst werden und die Konfiguration des Parsers wieder zurückgesetzt werden, um die äußere Sektion weiterzuverarbeiten. Der gesamte Parsing-Vorgang wird erfolgreich abgeschlossen, wenn die äußerste Sektion, welche die gesamte Datei darstellt,

korrekt verlassen werden kann.

6.6 Theoretischer Aufbau des Interpreters

Mit der vom Parser gelieferten Datenstruktur soll der Interpreter in der Lage sein, diese Daten sinnvoll grafisch darzustellen und deren Veränderung und Erweiterung zu erlauben. Weiterhin muss dann ein Ausgabemodul existieren, welches es ermöglicht, die Datenstruktur in der korrekten Syntax wieder zu speichern. Dieses Modul wurde noch nicht entwickelt, da es nach einem ähnlichen Prinzip arbeiten sollte wie der Parser.

Die Interpretation, also die Darstellung und Veränderung der Datenstruktur, erfolgt aus einer Kombination an festen Regeln und Heuristiken und den semantischen Informationen aus den Beschreibungsmodulen. Auch hier muss sichergestellt werden, dass diese Informationen ausreichen, um eine vollständige Interpretation zu ermöglichen. Als semantische Informationen existieren die Beschreibungen der einzelnen Sektionen und Optionen, die einfach nur an der richtigen Stelle darzustellen sind, aber vor allem auch die möglichen gültigen Optionen und deren struktureller Aufbau, woraus die gesamte Darstellung und die Veränderungsmöglichkeiten resultieren.

Die festen Regeln und Heuristiken bestimmen hauptsächlich, wie die semantischen Informationen verarbeitet werden und fangen eventuelle Standardfälle auf, die dann nicht explizit in allen Beschreibungsmodulen deklariert werden müssen.

Der Interpreter verwertet also die semantischen Informationen und bedient anhand der vom Parser gelieferten Datenstruktur über eine Schnittstelle die grafische Oberfläche. Hier ist es eventuell sinnvoll, zwischen dem Parser und dem Interpreter eine Umwandlung der Datenstruktur vorzunehmen, um ein leichtes, der Grammatik nahes Parsen zu ermöglichen, aber gleichzeitig ein optimales und unkompliziertes Interpretieren zu gewährleisten.

Da wir nie in die Implementierungsphase des Parsers gekommen sind, haben wir auch keine konkreten Implementierungsansätze für den Interpreter vorzuweisen. Wir sind allerdings der Meinung, dass, wenn die zugrundeliegende Datenstruktur gut ausgearbeitet ist, der Interpreter relativ leicht umsetzbar sein sollte.

7 Die Grafische Benutzerschnittstelle

7.1 Vorüberlegungen/Anforderungsprofil

Ein wesentliches Ziel von ConfyC war es, ein Programm zu entwerfen, mit dem ein Benutzer ohne spezifisches Wissen über die Syntax Konfigurationsdateien erstellen und modifizieren kann. Zu diesem Zweck bereiten wir, wie in den Kapiteln über Parser und Interpreter beschrieben, die eigentliche Konfigurationsdatei in einer für alle möglichen Konfigurationstypen gleichen Datenstruktur auf. Auf Basis dieser Datenstruktur wird dann dynamisch ein grafisches Abbild der Konfigurationsdatei erstellt, das sich aus Standard-Widgets wie ListBoxen, ComboBoxen, RadioButtons oder TextBoxen zusammensetzt.

Wie schon angesprochen, sind viele Konfigurationsdateien in einzelne Sektionen aufgeteilt, die semantisch zusammenhängende Optionen strukturieren. Diese Sektionen werden von der GUI ebenfalls grafisch hervorgehoben, so dass der Benutzer einen besseren Überblick über die Gesamtstruktur des Dokuments erhält.

Zur weiteren Unterstützung des Benutzers stellt ConfyC eine Online-Hilfe zur Verfügung. Dazu kann zu jeder Option der Konfigurationsdatei ein Hilfstext im Beschreibungsmodul gespeichert werden, der dem Benutzer sowohl Informationen zu der jeweiligen Einstellungsmöglichkeit als auch

sinnvolle Beispielwerte für diese Option liefert. Um auch Menschen mit nicht allzu ausgeprägten Englisch- bzw. Fremdsprachenkenntnissen eine Hilfe zu bieten, sollte es möglich sein, Texte in beliebigen Sprachen im Beschreibungsmodul zu speichern — dem Benutzer würde dann die Möglichkeit geboten, in einem Menü seine Wunschsprache, soweit vorhanden, auszuwählen.

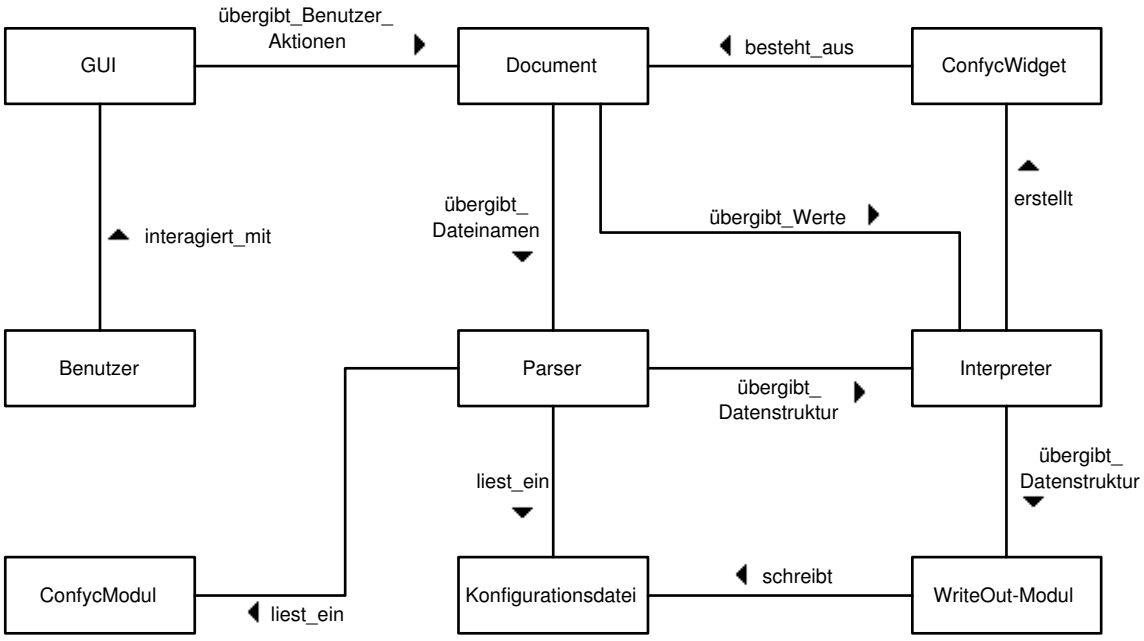
Die Oberfläche sollte nicht nur initial dynamisch aus einer Konfigurationsdatei und dem dazugehörigen Beschreibungsmodul generiert werden, auch während der eigentlichen Konfiguration durch den Benutzer würde sich die GUI beständig anpassen. Ein Beispiel dafür wäre das Deaktivieren einer ganzen Sektion — sinnvollerweise würde diese dann dem Benutzer auch nicht mehr angezeigt bzw. alle nicht mehr verfügbaren Optionen werden ausgegraut.

Bei der Benutzung von ConfyC muss zwischen drei verschiedenen Modi unterschieden werden. Zum einem bietet es die Möglichkeit, aus den in der Beschreibungsdatei vorhandenen Informationen eine initiale Konfigurationsdatei des gewünschten Typs zu generieren. Dem Benutzer werden dabei sinnvolle Default-Werte präsentiert, die er dann nach seinen eigenen Bedürfnissen anpassen kann. Die zweite Möglichkeit besteht im Öffnen und Bearbeiten einer bereits existierenden Konfigurationsdatei. ConfyC baut dabei die Widget-Struktur nach den vorhandenen Werten auf, die dann durch den Benutzer bei Bedarf geändert werden können. Der dritte Modus wäre die Einbindung eines externen Editors gewesen, so dass auch, wenn gewünscht, direkt der Quelltext der zu konfigurierenden Datei betrachtet oder geändert werden kann.

7.2 Modulinteraktion

7.2.1 Klassendiagramm

ConfyC gliedert sich in mehrere einzelne Module auf, die wir im folgenden Klassendiagramm dargestellt haben.



7.2.2 Modulbeschreibungen

Im Folgenden einige Erläuterungen zu den verwendeten Klassen und deren Bedeutung im Gesamtsystem:

Die Klasse *GUI* stellt die grafische Benutzerschnittstelle dar. Die eigentliche Darstellung der zur Konfiguration notwendigen *ConfycWidgets* wird in der Klasse *Document* gekapselt. Bei einem Objekt vom Typ *Document* handelt es sich also um die grafische Repräsentation einer Konfigurationsdatei, die dem Benutzer einfache Möglichkeiten zur Manipulation der Einstellungen und weitere Hilfestellungen bietet. Der *Parser* erhält den Pfad und Dateinamen sowie den Typ der zu konfigurierenden Datei und liest das passende ConfycModul, sowie die Konfigurationsdatei ein. Die vom *Parser* generierte Datenstruktur erhält der *Interpreter* um daraus die benötigten *ConfycWidgets* in einem *Document*-Objekt zu erstellen. *Document* und *Interpreter* interagieren dabei, da Änderungen, die der *Benutzer* vorgenommen hat, sowohl eine Neukonfiguration des *Documents* zur Folge haben können als auch diese Werte als Ergebnis an den Interpreter zurückgeliefert werden. Am Ende der Konfiguration übergibt der Interpreter dem *WriteOut-Modul* die modifizierte Datenstruktur, die von diesem zurück in die Konfigurationsdatei geschrieben wird.

7.3 Implementierung der GUI

7.3.1 Wahl des GUI-Toolkits

Wir hatten uns entschieden, ConfyC in C++ zu implementieren — als Kandidaten für die grafische Benutzeroberfläche boten sich deshalb GTK+[7] und Qt an. Mit beiden Toolkits wäre es sicher möglich gewesen, unser Vorhaben zu realisieren. Wir entschieden uns jedoch letzlich für Qt, da es im Gegensatz zu GTK+ objektorientiert ist, was im Zusammenhang mit C++ eine konsistentere Entwicklung ermöglichen sollte. Qt ist außerdem für viele verschiedene Plattformen, auch jenseits von UNIX/Linux verfügbar. So hätten wir ConfyC bei Bedarf ohne grosse Anpassungen zum Beispiel auch für MacOS X oder Windows umsetzen können. Da es sich bei ConfyC um freie und offene Software handeln sollte, gab es auch rechtlich kein Probleme, da Trolltech[8], der Hersteller von Qt, die Qt/X11 Free Edition unter die GPL[9] gestellt hat[10].

7.3.2 Verlauf der Implementierung

Da die grafische Benutzerschnittstelle neben dem Parser der aufwändigste Teil des ConfyC-Projektes war, entschlossen wir uns, diesen von Anfang an parallel zum restlichen Teil zu entwickeln. Dies erschien uns auch insofern als sinnvoll, da wir bis jetzt keine Erfahrungen mit Qt hatten und uns erst einmal vollständig in die Thematik einarbeiten mussten.

Zunächst überlegten wir uns, wie wir am elegantesten die geforderte dynamische GUI mit den Standard-Widgets von Qt umsetzen können. Dies erwies sich als nicht trivial, da einige Konfigurationsdateien einen sehr komplexen und verschachtelten Aufbau besitzen, aber durch die Oberfläche trotzdem klar strukturiert und übersichtlich dargestellt werden sollten.

Zum Anfang, auch um unsere Fähigkeiten in Qt auszubauen, implementierten wir den Rohbau der grafischen Benutzeroberfläche, genauer gesagt das Hauptfenster mit Menüstrukturen, Hilfesystem und der Signal/Slot-Infrastruktur von Qt. Als nächstes wurde begonnen, eine API zu entwerfen, mit deren Hilfe der Interpreter auf einfache Weise die Widget-Struktur der GUI anhand der Konfigurations- und Beschreibungsdatei aufbauen kann. Wie oben bereits erwähnt, wollten wir die Auswahlmöglichkeiten durch möglichst wenige Standard-Widgets realisieren, um die Bedieung möglichst einheitlich zu gestalten. Für Boolesche Optionen boten sich QRadioButtons an; waren relativ wenige Auwahlmöglichkeiten vorhanden, konnten sie gut durch QCheckBoxen oder QComboBoxen dargestellt werden. Für Werte, die vom Benutzer völlig frei gewählt werden können, wie zum Beispiel Namen oder beliebige Zahlenwerte, setzten wir auf QTextEdit-Felder. Teile dieser API wurden auch schon testweise implementiert — wir kamen dann jedoch nach einiger Zeit zu

[7] http://www.gtk.org/
[8] http://www.trolltech.com/
[9] http://www.gnu.org/copyleft/gpl.html
[10] http://www.trolltech.com/developer/licensing/

einem Punkt, an dem wir eine genaue Vorstellung davon haben mussten, wie der Interpreter arbeiten und welche Datenstruktur er vom Parser bereitgestellt bekommen würde, aus der letztendlich die GUI zusammengesetzt wird.

Durch die Schwierigkeiten, die wir bei der Realisierung des Parsers hatten, verzögerte sich so auch die Arbeit am Interpreter und somit die Implementierung der GUI. Als uns letztendlich klar wurde, dass wir zumindest im Rahmen dieser Lehrveranstaltung keinen lauffähigen Prototypen von ConfyC weden fertigstellen können, beendeten wir auch die Arbeit an der GUI.

8 Probleme

8.1 Technische Probleme

8.1.1 Probleme bei der Umsetzung des Parsers

Zur Implementierung des Parsers haben wir drei Möglichkeiten in Betracht gezogen. Zum einen kam das Verwenden eines schon vorhandenen, zumindest ähnlichen Parsers in Frage. Wir haben leider kein verwertbares Projekt gefunden.

Als zweites untersuchten wir die Möglichkeit, Parsergeneratoren zu verwenden. Beispiele hierfür sind *bison*[11] und *flex*[12] bzw. *bison++* und *flex++*, die C-Code, bzw. C++-Code liefern. Parsergeneratoren sind Programme, die anhand einer vorgegebenen Grammatik im günstigsten Fall den Code für einen Parser erzeugen. Im ungünstigsten Fall entsteht nur ein Rohbau, den man getrost auch von Hand schreiben kann. Parsergeneratoren verwenden die Grammatik zum Erstellen der Grundstruktur des Parsers und binden die möglichen Fallunterscheidungen ein. Zudem bieten sie die Möglichkeit, beliebig eigenen Code einzubinden, um auf bestimmte Ereignisse reagieren zu können. Die Verwendung von Parsergeneratoren scheitert allerdings in dem Punkt, dass unsere Grammatik keine konkreten Terminalsymbole verwendet. Damit kann der Generator keine eindeutigen Fallunterscheidungen erzeugen. Somit kann auch der Code für die Ereignisse, der verwendet werden sollte, die Rekonfigurationen auszulösen, nicht richtig platziert werden. Somit liefert der Parsergenerator im für uns günstigsten Fall nur einen Rohbau, der komplett zu überarbeiten ist, es sei denn, man kann das Problem der Terminalsymbole lösen. Hierzu wäre ein Ansatz, temporär einen Satz von disjunkten Symbolen zu verwenden, die später durch Variablen zu ersetzen sind, welche durch die Konfiguration belegt werden. Leider fehlte die Zeit, um diesen erst spät entwickelten Lösungsansatz überprüfen zu können.

Die dritte Möglichkeit — das komplette Neuschreiben des Parsers — unterscheidet sich kaum von der zweiten, da trotzdem eine Lösung für das Problem der nicht bekannten Terminalsymbole gefunden werden muss. Auch hier kann schnell ein Rohbau erstellt werden, der wiederum dann an besagtem Problem scheitert. In Hinsicht auf den Aufwand ist daher wohl das Verwenden eines Parsergenerators vorzuziehen.

Ein weiteres Problem ist die Rekonfiguration — zum einen die Konfiguration des Scanners, abhängig nur vom Dateityp, zum anderen die sektionsweise Rekonfiguration des Parsers während des Einlesens. Das Problem besteht hauptsächlich in der Koordination zwischen den Informationen, die das Beschreibungsmodul liefert und den Informationen, die während des Parsens gewonnen werden. Es empfiehlt sich, zuerst den Parser ohne die geforderten dynamischen Eigenschaften zu erstellen, um einen Überblick zu erhalten, wo die dynamischen Elemente einzubringen sind. Dafür muss aber die Struktur der Beschreibungsmodule feststehen, also vorher gesichert sein, dass die Struktur ausreicht, die Informationen aufzunehmen, die später für Parser und Interpreter notwendig sind. Ob alle diese Informationen enthalten sind, stellt sich aber wahrscheinlich erst im

[11] http://www.gnu.org/software/bison
[12] http://www.gnu.org/software/flex

Zuge der konkreten Implementierung des Parsers und Interpreters heraus, wodurch dann wiederum Veränderungen in der Struktur der Beschreibungsmodule resultieren können, womit die Basisimplementierung des Parsers verändert werden müsste. Daraus resultiert ein langwieriger zyklischer Softwareentwicklungsprozess. Hierbei ist das Verwenden eines Parsergenerators nur im ersten Zyklus anwendbar. Alternativ kann versucht werden, eine Methode zu finden, die Beschreibungsmodulstruktur dahingehend zu überprüfen, ob alle notwendigen Informationen aufgenommen werden können.

Weiterhin muss untersucht werden, ob jeweils eine eindeutige Rekonfiguration möglich ist. Die Basis für die Entscheidung, wie der Parser konfiguriert wird, ist die Information aus dem Beschreibungsmodul, aber sie wird erst einsetzbar, wenn während des Einlesevorgangs gesammelte Informationen verwertet werden. Problematisch gestaltet sich hier das dynamische Auffinden der richtigen Information, um eine korrekte und eindeutige Konfiguration des Parsers zu gewährleisten. So müssen beispielsweise Anfänge von Sektionen eindeutig sein. Welche und wie viele Informationen dafür notwendig sind, kann aber in den einzelnen Konfigurationsdateitypen unterschiedlich sein. Deshalb muss untersucht werden, ob sichergestellt ist, dass die Grammatikelemente eine eindeutige Unterscheidung ermöglichen, denn diese sind so definiert, dass sie immer mit der gleichen Art Daten belegt werden, um eben diese Unterscheidung zu ermöglichen. Leider sind uns keine Methoden bekannt, um die genannten Überprüfungen vorzunehmen, und es gibt daher nur die zeitaufwendige Möglichkeit des Ausprobierens.

8.2 Kommunikations- und Koordinationsprobleme

8.2.1 Arbeitsteilung

Als ein wesentliches Problem und wahrscheinlich auch ein Hauptgrund für das Scheitern des Projektes hat sich im Nachhinein die Aufgabenverteilung herausgestellt. Zum einen lag diese nämlich zu keinem Zeitpunkt in einer wirklich klaren Form niedergeschrieben vor, zum anderen war sie wohl auch nicht sonderlich gut durchdacht, wie wir leider erst feststellten, als sich schon einige Probleme daraus ergeben hatten.

Doch zunächst einmal zur Entwicklung der Aufgabenverteilung, wie sie sich im Verlaufe des Projekts ergab:

Zu Beginn war ein beträchtlicher Teil an theoretischer Arbeit zu leisten, welche sich vor allem in Form der Entwicklung der Grammatik widerspiegelte. Da sich Frank in der Vergangenheit schon recht häufig und ausgiebig mit Grammatiken beschäftigt hatte, wurde ihm diese Aufgabe übertragen.

Eine weitere anfänglich zu erledigende Aufgabe war die Einrichtung einer Internetpräsenz, die Jan-Ole in der Form übernahm, indem er uns bei *Sourceforge* anmeldete und auch die Website für das Projekt erstellte.

Thomas und Matthias hingegen hatten im Frühstadium des Projekts keine festen Aufgaben übernommen, haben sich jedoch beide in Qt eingearbeitet, da schon relativ früh feststand, dass die grafische Benutzerschnittstelle damit verwirklicht werden sollte und zuvor keiner damit gearbeitet hatte.

Nachdem die Grammatik komplett ausgearbeitet war, wurden die Aufgaben zur eigentlichen Implementierung verteilt. Da Frank die Grammatik und die theoretischen Grundlagen alleine erarbeitet hatte, sollte er die Implementierung von Scanner und Parser übernehmen. Thomas und Matthias, die sich, wie schon erwähnt, in QT eingearbeitet hatten, sollten ihre Kenntnisse dann für die Umsetzung der grafischen Benutzerschnittstelle einsetzen. Wir schätzten den Aufwand hierfür als gering genug ein, um nur einem der beiden diese Aufgabe zuzuweisen. Die Wahl fiel auf Tho-

mas, da dieser sich intensiver mit Qt beschäftigt und auch schon einigen Code geschrieben hatte.

Da nur noch der Interpreter als weiteres zu implementierendes Modul verblieb, wurde die Arbeit an diesem zweigeteilt. Matthias sollte hierbei die Schnittstelle zur GUI übernehmen, während Jan-Ole die Programmlogik implementieren sollte.

8.2.2 Daraus resultierende Probleme

Wie bereits erwähnt, ergaben sich aus dieser Aufgabenverteilung nun einige Probleme.

Als Fehler erwies sich, dass wir uns zu Beginn nicht alle an die theoretische Vorarbeit gesetzt haben, sei es nun gemeinsam oder unabhängig voneinander. Aufgrund der zuvor nicht absehbaren Komplexität der Aufgabe, nahm die Entwicklung der Grammatik erheblich mehr Zeit in Anspruch als vorgesehen. In diesem Zeitraum wurde einiges an potentieller Arbeitszeit aus zweierlei Gründen verschwendet. Einerseits arbeitete Frank hauptsächlich abends und zu Hause an der Grammatik, andererseits versäumten es die anderen drei, sich frühzeitig auch mit der Problematik zu beschäftigen. Es wäre auf jeden Fall sinnvoller gewesen, wenn sich alle zumindest Gedanken um die Theorie gemacht hätten. Zusätzlich musste dann Zeit investiert werden, sich in die Grammatik einzuarbeiten, als diese schließlich fertig war. Diese Einarbeitung erwies sich als äußerst aufwändig, da das Verständnis der Grammatik ohne ausgiebige vorherige Beschäftigung mit den benötigten theoretischen Grundlagen nur schwer möglich war.

Bei der eigentlichen Implementierung ergaben sich ähnliche Probleme. Die ursprüngliche Planung, den Prototypen während der Semesterferien fertigzustellen, wurde dadurch behindert, dass alle Gruppenmitglieder zu unterschiedlichen Terminen ihren Urlaub geplant hatten. Dies fiel insbesondere dadurch ins Gewicht, dass die Aufgaben so verteilt waren, dass vieles auf die Arbeit anderer aufbaute. So konnte zum Beispiel die GUI nicht weiterentwickelt werden, da ihr die Schnittstelle zum Interpreter fehlte. Dieser wiederum konnte aber nicht begonnen werden, da die vom Parser gelieferte Datenstruktur noch nicht definiert war. Als dies erkannt wurde enstand zwischenzeitlich der Plan eine separate Datenstruktur für den Interpreter zu entwickeln und dann ein weiteres Modul zur Überführung der vom Parser gelieferten Datenstruktur in die andere zu programmieren. Da wir erwarteten, den Parser rechtzeitig realisieren zu können, um auch Interpreter und GUI bis zum Abgabetermin fertigzustellen, verwarfen wir diese Idee wieder.

Die Aufgaben wurden also größtenteils so ungeschickt aufgeteilt, dass jeder in irgendeiner Weise auf die Ergebnisse eines anderen warten musste und somit der gesamte Arbeitsprozess aufgehalten wurde. Sinnvoller wäre gewesen, vor der eigentlichen Implementierung die Schnittstellen und Datenstrukturen gemeinsam zu erstellen, um erst dann, darauf aufbauend, das komplette Programm zu schreiben.

Abgesehen von diesen recht konkreten Problemen hatten wir auch Schwierigkeiten, vernünftig zusammenzuarbeiten. Eine mangelhafte Kommunikation, die sich in „Aneinandervorbeireden“ und Missverständnissen äußerte und welche teilweise in Streitigkeiten über die Mailingliste ausartete, erschwerte den Entwicklungsprozess sehr.

8.3 Sonstige Probleme

8.3.1 Dezentrale theoretische Entwicklungsarbeit

Die Erarbeitung der theoretischen Grundlagen hatte den größten Anteil an unserer Projektarbeit. Das Zusammenspiel der einzelnen Komponenten, der Grammatik, des dynamischen Parsers, der Beschreibungsmodule erfordert ein hohes Maß an Koordination und Interaktion.

Der gegenseitige Einfluss der genannten Komponenten aufeinander wirft Probleme auf, die in einer auch räumlich nah beieinander arbeitenden Gruppe besser und effektiver erkannt und gelöst

werden können. Wir haben versucht, die Ausarbeitung der Grundlagen verteilt zu bewältigen und zu spät erkannt, dass diese Vorgehensweise ein Vorwärtskommen ungemein erschwert. Auch wurden durch die Entwicklung an verteilten Einzelproblemen manche Zusammenhänge spät oder gar nicht erkennbar, aber auch die bekannten Zusammenhänge sind während der Entwicklungsphase kaum ausreichend dokumentiert gewesen. Eventuell wären diese teilweise komplizierten Darstellungen in einer Gruppe besser und schneller umgesetzt worden. Es scheint von Vorteil, so lange eine enge zentrale Entwicklungsstrategie zu verfolgen, bis ein gut dokumentierter, fundierter, verwertbarer Kern vorliegt, an dem dann dezentral weiterentwickelt werden kann. Anhand der dann vorliegenden Erkenntnisse und Ergebnisse fällt auch eine Aufgabenverteilung wesentlich leichter.

8.3.2 Motivation der ursprünglichen Idee vs. Motivation für die Umsetzung des konkreten Programms

Als zusätzlich erschwerend kamen mit zunehmender Dauer des Projekts auch noch persönliche Motivationsprobleme innerhalb der Gruppe hinzu. Diese resultierten zum einen daraus, dass das Problem sich als reichlich komplexer als angenommen darstellte und aufgrund andere Verpflichtungen durch das laufende Semester nicht ausreichend Zeit vorhanden war, um dieser Komplexität gerecht zu werden, zum anderen aber auch sicherlich daraus, dass wir feststellten, dass wir höchstwahrscheinlich nach der Fertigstellung von ConfyC dieses gar nicht selber anwenden würden. Dieses Problem spiegelt sich vorzüglich in einem Zitat von Eric Raymond aus „The Cathedral and The Bazaar“ [13] wider:

> *1. Jede gute Software beginnt mit den persönlichen Sehnsüchten eines Entwicklers. Das hätte vielleicht jedem sofort einleuchten sollen (schließlich gibt es das Sprichwort „Not macht erfinderisch“ schon seit geraumer Zeit), aber viel zu oft haben Softwareentwickler ihre Tage mit der Arbeit an Programmen verbringen müssen, die sie weder gebraucht noch geliebt haben. Aber nicht in der Linux-Welt — was vielleicht die überdurchschnittliche Qualität der von der Linux-Gemeinde geschaffenen Software erklärt.“*

In unserem Fall stimmte dies nicht ganz. Zwar schien uns allen das Projekt von Anfang an interessant, aber von wenig persönlichen Nutzen. Dazu waren wir zu weit von der ursprünglichen Idee des *fvwm2*-Konfigurationstools entfernt, für welches zumindest ein Teil der Gruppe Verwendung hatte. Bis auf dieses Level zurückzugehen und statt ConfyC ein solches Tool zu entwickeln, empfanden wir aber trotzdem als falsch, da wir uns letztlich für das komplexere entschieden hatten und, davon abgesehen, bei dieser Erkenntnis die Zeit auch für das vereinfachte Problem zu knapp war. Im Nachhinein wird aber überdeutlich, dass auch diese fehlende Motivation ein entscheidener Grund für das Scheitern ist.

8.3.3 Suche nach verwandten Projekten

Ein weiterer Punkt, den wir vielleicht nicht ausreichend verfolgt haben, war die Suche nach ähnlichen bzw. verwertbaren Projekten. Sicherlich haben wir nach für uns verwendbaren Programmen wie z.B. Parsergeneratoren gesucht und diese als untauglich befunden, die Tatsache, dass wir allerdings das *COAS*-Projekt[14] nicht selber gefunden haben, sondern erst beim Workshop am Ende des Semesters darauf aufmerksam gemacht wurden, spricht dafür, dass wir bei der Suche wohl zu wenig Geduld aufgebracht haben. Zwar ist *COAS* als eigenständiges OpenSource-Projekt offensichtlich aufgegeben worden, zumindest datiert die letzte Änderung auf der COAS-Website vom September 1999 und der komplette COAS-FTP-Server ist nicht mehr erreichbar, jedoch scheint Caldera *COAS* weiter intern in ihrer Distribution einzusetzen, von der wir dann auch eine aktuellere Version erhalten haben. *COAS* ließ sich bei uns leider nicht zum Laufen bringen und erwies sich als sehr gross und wenig aufschlussreich dokumentiert — aber ein frühzeitiger Kontakt mit dem ursprünglichen Entwickler hätte uns vielleicht seine Probleme aufzeigen können, um dadurch

[13] http://www.phone-soft.com/RaymondCathedralBazaar/catb_g.2.html
[14] http://www.coas.org/

Rückschlüsse auf die Durchführbarkeit unseres Projektes ziehen zu können oder sogar Ansätze zu finden, die eine Realisierung von ConfyC doch möglich gemacht hätten.

Im Laufe dieses Fehlerberichts haben wir noch einige weitere Programme und Bibliotheken entdeckt, deren Websites darauf hindeuten, dass sie ein zumindest ähnliches Ziel wie ConfyC verfolgen. Aufgrund mangelnder Zeit konnten wir diese Programme aber nicht mehr testen.

Alles in allem lassen diese Ergebnisse den Schluß zu, dass eine genauere Suche nach verwertbaren Projekten eventuell die Arbeit hätte erleichtern können. Zu erwähnen ist aber auch, dass keines der gescheiterten Projekte ihre Probleme publik gemacht hat, so dass spätere Entwickler (wie wir) aus deren Fehlern und Problemen lernen können. Natürlich wäre aber auch, wenn wir frühzeitiger derartige Projekte gefunden hätten, ein E-Mail-Kontakt möglich und unter Umständen hilfreich gewesen.

8.4 Alternative Problemlösung

Dem Problem, welches ConfyC lösen soll, kann auch auf eine andere Weise entgegengewirkt werden als mit der Entwicklung und Fortführung unseres Projektes. Eine syntaktisch und semantisch einheitliche Struktur für Konfigurationsdateien ist eine alternative Lösung. Diese muss mächtig genug sein, alle Informationsgehalte aufzunehmen, die auch von heutigen und eventuell zukünftigen Konfigurationen benötigt werden. Des weiteren sollte dann eine Bibliothek entwickelt werden, welche die gesamten Verwaltungsroutinen beinhaltet. Selbstverständlich kann auch eine schon vorhandene Lösung dafür verwendet werden. Möglicherweise bietet sich hier XML an.

9 Schluß

Diese Arbeit sollte einerseits den Verlauf des Projektes aufzeigen und die erzielten Ergebnisse — insbesondere auch im Bereich der Theorie — darstellen, andererseits aber auch die gemachten Fehler nennen und analysieren. Insbesondere die Probleme waren vielfältiger und vor allem schwerwiegender, als wir zu Beginn der Entwicklung geglaubt hatten. Viele der Fehler waren, wie jetzt nach der Analyse deutlich wird, vollkommen unabhängig vom eigentlichen Projekt und sind somit auch übertragbar und als positive Erfahrung verwertbar in allen Bereichen der Gruppenarbeit. Davon abgesehen gab es natürlich auch große Probleme, die ein Projekt wie ConfyC direkt betreffen und bei denen diese Arbeit vielleicht anderen Entwicklern als Hilfestellung dienen kann.

Aber gerade in Bezug auf die die Gruppenarbeit betreffenden Probleme erscheint es umso erstaunlicher, dass wir bei der Ausarbeitung dieses Abschlussberichtes viele der Fehler, die wir bei der eigentlichen Projektarbeit gemacht hatten, nun vermieden haben. Somit zeigt alleine die Beschäftigung mit dieser Fehleranalyse einen Sinn „aus sich selbst heraus“, denn mit ihr haben wir die Möglichkeit genutzt, die erlangten Erfahrungen einzusetzen.

10 Anhang

Website von ConfyC: http://confyc.sourceforge.net
ConfyC@sourceforge: http://www.sourceforge.net/projects/confyc
DESE-Homepage: http://flp.cs.tu-berlin.de/dese/