Entwurf und Implementierung der Steuerung eines Temperaturkontrollsystems

Georg Stegmüller
8630572

Juli 1998

Diplomarbeit in Wirtschaftstelematik

durchgeführt am

Institut für Softwaretechnologie
der Technischen Universität Graz

Begutachter: O. Univ. Prof. DI Peter Lucas
Betreuer: DI Brigitte Fröhlich

Danksagung

Diese Diplomarbeit entstand in Zusammenarbeit mit dem *Institut für Softwaretechnologie* der Technischen Universität Graz und dem *Institut für Geologie und Paläontologie* der Karl Franzens Universität Graz. Die Betreuung seitens der KFUG erfolgte durch Mag. Jürgen Loizenbauer. Mein besonderer Dank gebührt Fr. DI B. Fröhlich vom Institut für Softwaretechnologie, die mir immer mit Rat und Tat zur Seite stand.

Zu Dank verpflichtet bin ich vor allem meiner Mutter und meiner Familie für die Unterstützung, die ich von Ihnen während meiner Studienzeit empfangen habe.

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am ....
(Unterschrift)

ZUSAMMENFASSUNG

Das Institut für Geologie und Paläontologie an der Grazer Karl Franzens Universität führt Untersuchungen an Flüssigkeitseinschlüssen in Gesteinen durch. Um den Experimentator dabei zu unterstützen, ist ein Steuerprogramm für einen Front-End-Rechner zu entwickeln.
Zentrale Elemente dieser Diplomarbeit:

- Aus den intuitiven Erwartungen und Vorstellungen die wahren Anforderungen an das System herauszukristallisieren, zu konkretisieren und schlußendlich zu definieren.
- Aufgaben zu finden, die bis jetzt von Hand gemacht wurden, obwohl der Computer sie übernehmen kann (d.h. das bestehende System zu erweitern) und weiters Erwartungen, die an den Computer gestellt werden, auf ihre Sinnhaftigkeit zu überprüfen (z.B. die graphische Eingabe von Temperaturen).

Ergebnis dieser Diplomarbeit ist zum einen die systematische Analyse und Spezifikation der Requirements und zum anderen deren Umsetzung in einer Implementation. Außerdem soll der Entwicklungsprozeß beobachtet werden.
Leider stand mir nur ein eingeschränkter Spielraum zur Verfügung, da einige Entscheidungen bereits vor meinem Arbeitsbeginn getroffen wurden (z.B. war ein Videoprinter und ein eigener Monitor bereits geordert, ansonsten hätte man eine Framegrabberkarte oder ähnliches einsetzen können).
Die eigentliche Herausforderung bestand darin, daß die intuitiven Erwartungen und Vorstellungen naturgemäß nur sehr diffus vorhanden waren.

INHALTSVERZEICHNIS

TABELLENVERZEICHNIS

ABBILDUNGSVERZEICHNIS

1. ALLGEMEINES ZUR REQUIREMENTSANALYSE

1.1 Was sind Requirements?

Requirements sind Eigenschaften bzw. Funktionen, die von der Hard/Software erfüllt werden müssen. Wenn auch nur eines davon nicht so vorhanden ist, wie es spezifiziert wurde, gilt das Projekt als gescheitert. "Requirements" kann in diesem Zusammenhang am besten mit "Anforderungen" übersetzt werden. Trotz dieser guten Übersetzung werde ich weiter "Requirements" verwenden, da der Hauptteil der Literatur zu diesem Thema auf englisch vorhanden ist.

1.2 Warum benötigt man die Requirementsanalyse?

Aus der Sicht der Softwarehersteller ist die gegenwärtige Lage ein Paradies. Heute verkaufte Software garantiert oft nicht einmal jene Funktionen, von denen der Hersteller behauptet, daß das Programm sie erfüllt.

Für den (Groß-)Kunden ist diese Situation auf die Dauer untragbar (und ich finde es faszinierend, daß diese Situation überhaupt entstehen konnte). Es wird daher zu einem entscheidenden Wettbewerbsvorteil werden, bei einem Softwareprojekt Garantien bieten zu können. Dafür ist es aber nötig, eine genaue Beschreibung des Umfangs und der Fähigkeiten der Software zu erstellen, um eine Referenz zu erhalten. Dies ist die Aufgabe der Requirementsanalyse. Darüber hinaus bietet sie auch die Grundlage, eine Kostenabschätzung vornehmen zu können.

1.3 Wie geht man nun vor?

1.3.1 Systemanalyse

Der erste Schritt ist die Systemanalyse. Ihr Zweck ist das Kennenlernen jeder Facette jenes Systems, das (zumindest in Teilbereichen) durch EDV ersetzt oder verbessert werden soll. Die Forderung "Alle Aspekte des Systems zu analysieren" ist in der Praxis ob der Komplexität der Abläufe in der Regel unrealistisch. Daher liegt es im Aufgabenbereich des Analytikers, die *wichtigen Aspekte* zu identifizieren und sich bei der Analyse auf diese zu beschränken. Die Analyse selbst wird durch eine große Zahl von formalen Methoden unterstützt. [Dav90] gibt eine umfangreiche Gegenüberstellung von verschiedenen Methoden. Im wesentlichen gibt es zwei grundsätzliche Ausgangspunkte:

1. Datenflußorientiert
2. Objektorientiert

Für jede dieser Analysephilosophien gibt es eine Vielzahl von verschiedenen Methoden und Variationen. Es wäre ein Fehler, sich eine Methode heraus zu suchen, und diese überall anzuwenden. In [Dav90] wird der Vergleich mit einem Handwerker und seinem Werkzeugkasten verwendet. Der geschickte Analytiker kennt viele Verfahren und sucht das für das aktuelle Projekt geeignetste heraus, wie auch der geschickte Handwerker nicht einen Hammer verwendet, um eine Schraube in die Wand zu bekommen (obwohl es funktionieren kann).

Allen diesen Methoden ist eines gemein: Mit möglichst vielen beteiligten Personen sprechen, und die (unterschiedlichen) Sichtweisen zu vereinen.

Ergebnis der Systemanalyse ist das *vollständige* Verstehen des Problems bzw. seiner wichtigsten Aspekte. Dazu werden umfangreiche Diagramme angefertigt, die in den nachfolgenden Schritten als Basis für Entscheidungen dienen.

1.3.2 Definition der Requirements

Spätestens zu diesem Zeitpunkt wird entschieden:

- Welche Aufgaben können von der EDV übernommen werden?
- Welche Aufgaben sollen von der EDV übernommen werden?
- Welche Projektgröße kann finanziert werden?

Wenn der äußere Umfang bestimmt ist, werden die Requirements gesucht. Damit ist gemeint, welche Teilaufgaben gelöst werden sollen, welches Format die Eingaben und Ausgaben besitzen sollen, ob, und wenn ja, welche Zeiten einzuhalten sind etc. Erstaunlicherweise wird über diese Phase wenig gesprochen. Dabei steht und fällt die Nutzbarkeit des neuen Systems mit ihr. Requirements, die hier nicht erkannt werden, haben es äußerst schwer, in das System aufgenommen zu werden und Requirements, die nicht unbedingt nötig sind, aber später große Kosten verursachen, können hier sehr einfach eliminiert werden.

In dieser Phase tritt aber ein grundsätzliches Problem auf. Diejenigen Personen, die beurteilen können, welche Funktionen notwendig sind (in der Regel die End-User), sind sehr oft nicht in der Lage, zu beurteilen, welche Funktionen überhaupt möglich sind. Daraus folgt, das einige wünschenswerte Funktionen nicht in die Spezifikation aufgenommen werden, da angenommen wird, sie seien unerfüllbar.

Umgekehrt gibt es dieses Problem ebenso. Viele mögliche Funktionen werden nicht erwähnt, da die Auffassung (Befürchtung) besteht, sie seien nur eine Spielerei, und deshalb nicht notwendig.

Deshalb ist eine gute Zusammenarbeit zwischen End-Usern und Programmierern notwendig. Es gilt dabei, folgende Fragen zu beantworten:

1. "*Was kann vom Programm durchgeführt werden?*"
2. "*Was soll vom Programm durchgeführt werden?*"

Die erste Frage ist notwendig, um eine möglichst vollständige Wunschliste der End-User zu ermöglichen. Die zweite Frage soll ein Zuviel an Funktionen verhindern, welche nur die Bedienbarkeit und auch die Erstellung des Programms erschwert.

1.3.3 Spezifikation der Requirements

Die Punkte, die in der vorhergegangenen Phase gesammelt wurden, müssen nun genau spezifiziert werden. Die Spezifikation muß genau und eindeutig sein, um einerseits den Programmierern eine eindeutige Richtlinie zu geben und andererseits, um eine nachträgliche Überprüfung auf (insbesondere qualitative) Erfüllung der Requirements zu ermöglichen. Bei Nichterfüllung ist es dann relativ einfach, den Verursacher zu finden. Es kann sehr einfach gezeigt werden, ob falsch implementiert wurde, ob falsch "designed" wurde oder ob unpassende Funktionen verlangt wurden.

Trotz der genauen Spezifikation sollen aber nach Möglichkeit keine Implementierungsdetails vorweggenommen werden. Im Gegensatz zur Requirements Definitionsphase stehen für diesen Abschnitt des Entwicklungsprozesses wieder eine Reihe von formalen Hilfsmitteln zur Verfügung [Dav90]. Interessanterweise stammen viele dieser Verfahren aus dem Bereich von Telefongesellschaften. Ein anderer wichtiger Bereich ist die formale Spezifikation von Programmiersprachen [BJ78].

Vielleicht ist das der Grund, daß die Regeln für die Erstellung von GUIs sehr einseitig sind. Man findet zum Beispiel in [vFH91] eine Reihe von Regeln zum grundsätzlichen Design von Dialogen und Fenstern und deren Verhaltensmuster. Es wird eine grundsätzliche Anleitung zur Benutzerführung gegeben, die praktisch für jede Anwendung richtig ist. Weiters gibt es umfangreiche Regelwerke (z.B. von IBM) wie die Oberfläche eines Programms auszusehen und auf Usereingaben zu reagieren hat. Damit wird erreicht, daß das Wissen über die Bedienung eines Programmes bei allen anderen Programmen auf dieser Plattform (zumindest in großen Teilen) wiederverwendet werden kann. Es wird daher definiert, in welcher Reihenfolge bestimmte Menüeinträge zu finden sind, welchen Namen sie besitzen, in welcher Art und Weise die Druckknöpfe in Dialogen angeordnet und gezeichnet werden müssen und ähnliches.

Was man leider nicht findet, ist ein analytischer Ansatz zur Klärung von Fragen wie: "Welche Information soll überhaupt dargestellt werden?" bzw. "Wie soll eine bestimmte Information dargestellt werden?" Dieser Zustand ist insoferne bedauerlich, da die Nützlichkeit eines Programms sehr wohl mit der Qualität der Informationsdarstellung (und daher auch der Filterung von Information) zusammenhängt. Daß die Formalisierung dieses Bereiches des Designs nicht einfach ist, erkennt man an diversen SIGCHI-Bulletins zu diesem Thema ([RU93][Bil93c][Bil93b] [Wix93]). Die europäische Gemeinschaft nimmt sich auch dieses Problems an ([Bil93a]).

Bis dahin muß man ohne einen formalen Leitfaden ein geeignetes GUI entwicklen und kann eigentlich nur mehr die Methode "Versuch und Irrtum" anwenden. Unter Mitarbeit einer Anzahl von End-Usern werden Prototypen des endgültigen Interfaces erstellt und in mehreren Zyklen verfeinert. Während diese Strategie den Vorteil hat, daß sie zum einen relativ schnell ziemlich gute Ergebnisse liefert, hat sie den Nachteil, einen einmal eingeschlagenen Weg nicht mehr leicht verlassen zu können. Außerdem sind die Tester durch die wiederholten Testzyklen einem Lernprozeß unterworfen, der die Testergebnisse bezüglich der Einfachheit der Bedienung verfälscht.

1.3.4 Implementation / Tests

Sehr gerne übersehen wird die Phase der Implementierung und mit ihr die Testphase. Da Probleme, die erst hier erkannt werden, dazu führen, daß der Prozeß der Definition und Spezifikation von Requirements zumindest teilweise neu durchlaufen werden muß, ist es im Sinne einer Kostenreduktion und Beschleunigung der Fertigstellung sinnvoll, die Phasen nicht strikt getrennt, sondern überlappend durchzuführen.

1.3.5 Einführung / Schulung / Wartung

Wenn das Programm fertiggestellt wurde, endet der Prozeß meist nicht, sondern muß durch fortlaufende Wartung, Einschulung von Personal und ähnlicher Zusatzleistungen zumindest für einige Zeit fortgeführt werden. Es kann daher sinnvoll sein, auch für diese Aufgaben Requirements zu erstellen und zu spezifizieren.

1.4 Bemerkungen zur Diplomarbeit

Diese Diplomarbeit war insoferne atypisch, als die Analyse, Spezifikation und Implementierung nur von einer Person durchgeführt wurden. In der Phase der Definition der Requirements war sie allerdings beispielhaft, da der Umfang und die Art der EDV-Lösung anfangs nur sehr ungenau bekannt waren. Die hervorragende Zusammenarbeit zwischen End-User und Designer hat dazugeführt, daß die EDV-Lösung genau auf die Wünsche und Notwendigkeiten der End-User eingeht, ohne ein Zuviel an Funktionen aufzuweisen, die nie benötigt werden und nur den Umfang vergrößern, während sie die Bedienbarkeit vermindern.

Die Zielsetzung des TCS (Temperature Control System) ist intuitiv sehr schnell klar. *Das TCS soll den Untersuchenden unterstützen.* Auch waren die großen Aufgaben, wie das Eingeben und Abarbeiten eines Temperaturverlaufes oder die Erstellung und Verwaltung der Ereignisliste, von Anfang an *relativ* genau umrissen. Es war jedoch nicht von Anfang an klar, wie diese Aufgaben erfüllt werden sollen bzw. welche Detailaufgaben daraus abgeleitet werden sollen.

Auch hat sich der Umfang des Projekts (in Grenzen) während der Arbeit erweitert, hauptsächlich aus zwei Gründen:

1. Dadurch, daß bisher nur mit einem speziellem Temperatur-Controller der Firma Omega (deshalb auch Omega-Controller genannt) gearbeitet wurde, war einigermaßen bekannt, was an dieser Lösung schlecht war (z.B. Eine Meßreihe durchführen, und danach von Hand kalibrieren, oder das Aufnehmen der Meßwerte an sich). Es gab aber keine Überlegungen, welche Aufgaben das TCS zusätzlich *neu* übernehmen könnte. Einige solcher Aufgabengebiete finden sich ab Seite 75.

2. Sobald bestimmte Funktionen von einem Computer übernommen werden, kann man ihn auch dazu verwenden, diese Funktionen zu erweitern. So bietet zum Beispiel der Omega-Controller die Möglichkeit, ihn relativ spartanisch zu programmieren. Da nun der Omega-Controller aber vom TCS gesteuert wird, war es ein leichtes, Funktionen hinzuzufügen (z.B. die Benachrichtigungsbefehle).

Leider konnten einige interessante Ansätze nicht verwirklicht werden. So gab es zu Beginn des Projekts die Idee, den Temperaturverlauf graphisch einzugeben. Die Idee entstand im wesentlichen wohl deshalb, da die Temperaturverläufe auch immer graphisch *dargestellt* werden. Diese an sich gute Idee konnte aber nicht realisiert werden, da Temperaturveränderungen in ihrer Reaktionszeit relativ komplex und von vielen Faktoren (Tunesets, Temperaturdifferenz, Umgebungstemperatur, Ausgangstemperatur etc.) abhängig sind. Außerdem ist die genaue Eingabe eines Temperatur/Zeitpaares mit der Maus verhältnismäßig umständlich.

Aus diesen Umständen heraus ergibt sich auch meine Forderung nach Einbeziehung von Vertretern der End-User als auch von Personen, die mit der Implementation entweder tatsächlich zu tun haben, oder zumindest das Zielsystem gut kennen. Letztere sind vor allem deshalb notwendig, um Wünsche in realisierbar, aufwendig realisierbar oder gar nicht (bzw. sehr schlecht) realisierbar einzuteilen.

Wie sich Requirements im Laufe der Zeit ändern können, möchte ich am Beispiel der Profile-Sprache erläutern:

Der Omega Controller kann programmiert werden. Die Programme nennt man "Profile". Der Befehlssatz ist recht spartanisch, aber der Aufgabe durchaus angemessen. Das User Interface (UI) zur Programmierung ist aber unangenehm, was zum Teil auf die Konstruktion des Controllers zurückgeführt werden kann (kleine Anzeige, Tasten an der Front des Gehäuses, etc.).

- Eine der Aufgaben des TCS ist, den Temperaturverlauf eingeben zu können. Da gleich am Anfang die graphische Eingabe als unpraktisch abgewiesen wurde, war ursprünglich beabsichtigt, einen *Crosscompiler* zu

entwickeln. Die am Computer eingegeben Befehle werden also in die Kommandos übersetzt, die der Controller versteht, danach werden diese zum Controller übertragen und das TCS wartet, bis der Controller sein Programm abgearbeitet hat.

- Bei der Durchsicht der Beispiele und im Gespräch wurde deutlich, daß sich bestimmte Teilaufgaben ähnlich sind. Da der Controller zwar Unterprogramme kennt, aber keine Ahnung von Variablen hat, entstand die Idee zu einem *Präprozessor*.

- Da während eines Experiments Phasen vorhanden sind, die lange dauern aber ereignislos sind (z.B. kann das Abkühlen am Beginn des Experiments mehr als 20 Minuten dauern), wurden Benachrichtigungsbefehle ersonnen. Da der Omega Controller aber nicht in der Lage ist, akustische Feedbackereignisse für den Benutzer zu generieren, sollte der Programmfortschritt des Omega Controllers durch das TCS überwacht werden und zu den beabsichtigten Zeiten sollte das TCS das passende Feedback generieren.

- Da dieser Ansatz aber sehr aufwendig und trotzdem ungenau war, entschloß ich mich, die Profileabarbeitung in das TCS zu integrieren. Dieser Zeitpunkt war die Geburtsstunde der *Profilesprache* und einer Mehrdeutigkeit in der Bezeichnung. Das Programm, welches auf dem Omega Controller läuft, wird als "Profile" bezeichnet. Da das TCS für den Anwender transparent sein soll, heißt das Programm im TCS ebenfalls "Profile". In der Folge werde ich jedesmal, wenn ein Profile des Omega Controllers gemeint ist, dieses genau angeben. Wenn hingegen nur von einem "Profile" die Rede ist, ist jenes vom TCS gemeint.

Anzumerken ist folgendes: All diese Entwicklungsschritte wurden von mir (d.h. dem Entwickler) vorgeschlagen. Ich denke, daß diese Situation aber typisch ist und meiner Meinung nach gibt es dafür folgende Gründe:

1. Der Anwender weiß nicht, daß er einen bestimmten Wunsch hat. Das heißt nicht, daß der Anwender dumm ist, er nur nie auf die Idee gekommen, einen Vorgang anders zu machen.

2. Der Anwender weiß um den Wunsch, glaubt aber er sei unrealisierbar, zu aufwendig oder einfach lächerlich.

3. Der Anwender geht davon aus, daß der Wunsch selbstverständlich ist.

Um diese 3 Punkte in den Griff zu bekommen, ist bei den Besprechungen immer wieder darauf hinzuweisen, die bestehende Lösung zu vergessen und sich eine Wunschlösung vorzustellen. Ich hoffe, daß ich mit meiner Arbeit diesem Ideal entspreche.

2. SPEZIELLE AUFGABENSTELLUNG

2.1 Experimentbeschreibung

Der Experimentator verwendet zur Untersuchung von Flüssigkeitseinschlüssen in Gesteinsdickschliffen ein Durchlichtmikroskop. Um die natürlichen Flüssigkeiten dabei zu bestimmen, wird das Phasenverhalten dieser Fluide unter verschiedenen Temperaturen beobachtet. So ist zum Beispiel das Schmelzen von eingeschlossenem Eis bei -56.6 Grad C ein Hinweis auf ein reines CO2 Fluid. Wenn eine wasserreiche Phase unter 0 Grad C schmilzt, weist dies auf gelöste Salze hin.

Zur Untersuchung wird nun die Probe in einen Heiz-/Kühltisch eingespannt. Danach legt der Experimentator einen Temperaturverlauf fest, der ihm interessant erscheint. Während dieser Temperaturverlauf am Heiz-/Kühltisch erzeugt wird, beobachtet der Experimentator die Probe. Jedesmal, wenn sich die Probe verändert (z.B. wenn ein Flüssigkeitseinschluß schmilzt), notiert der Experimentator die Temperatur des Heiz-/Kühltisches und kennzeichnet diese Veränderung in einer Zeichnung oder macht von der Probe ein Bild.

Das Endergebnis des Experiments ist zum einen eine Liste von diesen protokollierten Veränderungen und ein oder mehrere Bilder der Probe. Für ein Beispiel siehe auch Seite 80.

2.2 Situation vor der Einführung des TCS

2.2.1 Geräteaufbau

2.2.1.1 Omega Controller

Die eigentliche Aufgabe der Temperatursteuerung übernimmt der Omega CN-3000 Controller. Dafür verwendet er zwei getrennte Ausgänge: Ein Ausgang steuert den Heizvorgang, der andere den Kühlvorgang. Für jeden dieser Ausgänge gibt es eine eigene PID-Steuerung (siehe Seite 79). Jeweils ein PID-Triple für den Heiz- bzw. Kühlausgang bilden ein sogenanntes Tuneset.

Der Controller kann auf zwei Arten betrieben werden:

1. Der Benutzer gibt einen Sollwert für den Heiz-/Kühltisch an. Der Controller wird sofort beginnen, die Ist-Temperatur an die Soll-Temperatur anzugleichen. Dieser Vorgang wird nur durch das aktuelle Tuneset beeinflußt.

2. Der Benutzer programmiert den Controller. Es stehen Plätze für bis zu 8 Tunesets zur Verfügung, die einzeln aktiviert werden können. Zur Temperatursteuerung stehen spezielle Anweisungen zur Verfügung, die neben einer Toleranzgrenze auch eine Zeitvorgabe beinhalten.

In Abhängigkeit der PID-Werte und der Abweichung zwischen Ist- und Soll-Temperatur ermittelt der Omega Controller Steueranweisungen für den Chaimexa 2-Punktregler.

Über eine RS232-Verbindung kann der Omega Controller von einem PC ferngesteuert werden.

2.2.1.2 Chaimexa 2 Punktregler

Da der Omega Controller die Steuersignale berechnet, ist die Funktion dieses Reglers auf die Relaissteuerung beschränkt. Zum Heizen wird eine 24V Spannung an einen Heizdraht gelegt, zum Kühlen wird ein Ventil einer Gasflasche geöffnet. Aus der Sicht des Users bilden Omega Controller und Chaimexa Regler eine logische Einheit.

2.2.1.3 Mikroskop

Das Mikroskop ist ein Durchlichtmikroskop der Marke Olympus Typ PH2.

2.2.1.4 Computer

Hardware

Als Computer steht ein AT-Kompatibler 486DX-PC zur Verfügung. Er besitzt 4 MB Hauptspeicher. Die Graphikkarte besitzt einen TRIDENT-Chipsatz und stellt 800x600 Punkte in 16 Farben zur Verfügung. An dieser Karte wird ein Farbmonitor betrieben.

Software

Als Betriebssystem wird MSDOS 6.0 und MS Windows 3.1 verwendet.

2.2.1.5 Sonstiges

1. Panasonic WVBL 600 CCTV-Kamera

2. Hitachi Videorekorder

3. Mitsubishi Video Copy Prozessor P67E - Videoprinter
4. SW-Monitor

2.2.2 Systemanalyse

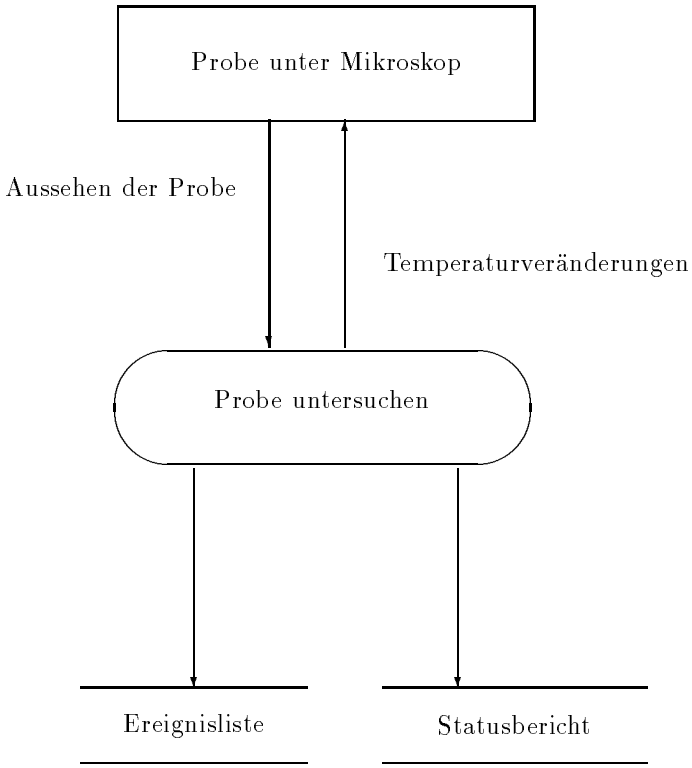

Abb 1 Experiment-Überblick

Abb. 1 gibt einen Überblick über das Experiment. Zentraler Vorgang ist das Untersuchen der Probe. Dafür wird die Probe unter dem Mikroskop beobachtet und die Temperatur des Heiz/Kühltisches des Mikroskops wird verändert. Diese Temperaturveränderungen werden in einem Statusbericht mitprotokolliert. Jedesmal, wenn sich die Probe entscheidend verändert (z.B. ein gefrorener Flüssigkeitseinschluß schmilzt), ist dies ein Ereignis, welches in der Ereignisliste vermerkt wird.

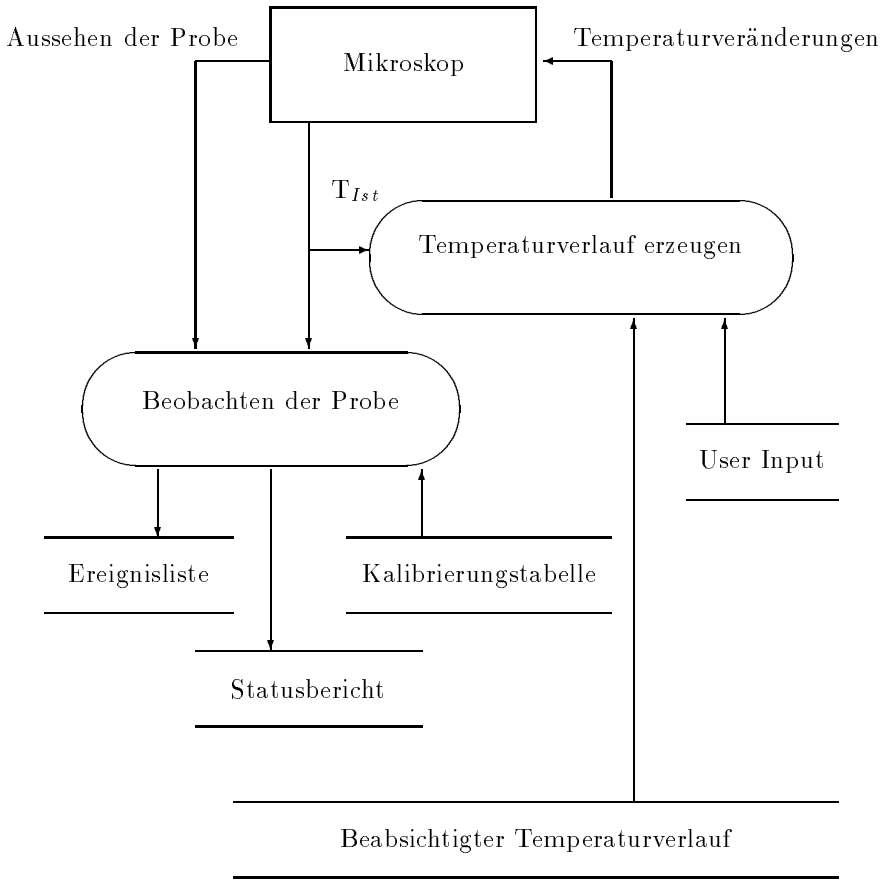

Abb 2 Experiment - Detail

Abb. 2 beschreibt das Experiment genauer. Hervorzuheben ist die Trennung des Vorgangs *Probe untersuchen* in die beiden Einzelvorgänge *Beobachten der Probe* und *Temperaturverlauf erzeugen*.

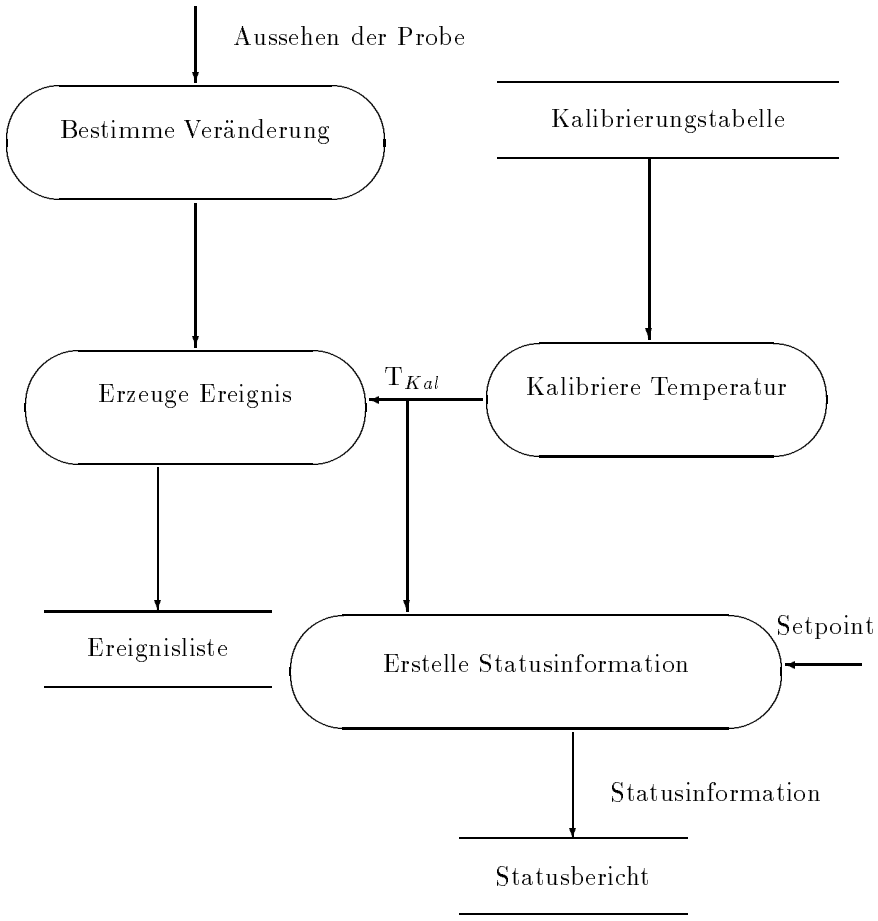

Abb 3 Probe untersuchen

Abb. 3 stellt den Vorgang *Beobachten der Probe* genau dar. Dieser zerfällt in folgende Einzelvorgänge:

1. Bestimme Veränderung: Es wird festgestellt, ob eine Veränderung eingetreten ist und wenn ja, welcher Art sie ist.

2. Erzeuge Ereignis: Aus einer Veränderung und der kalibrierten Temperatur wird ein Ereignis erzeugt.

3. Kalibriere Temperatur: Die vom Heiz/Kühltisch berichtete Temperatur wird kalibriert.

4. Erstelle Statusinformation: Solltemperatur und kalibrierte Ist-Temperatur und die Zeit werden zu einer Statusinformation zusammengefaßt.

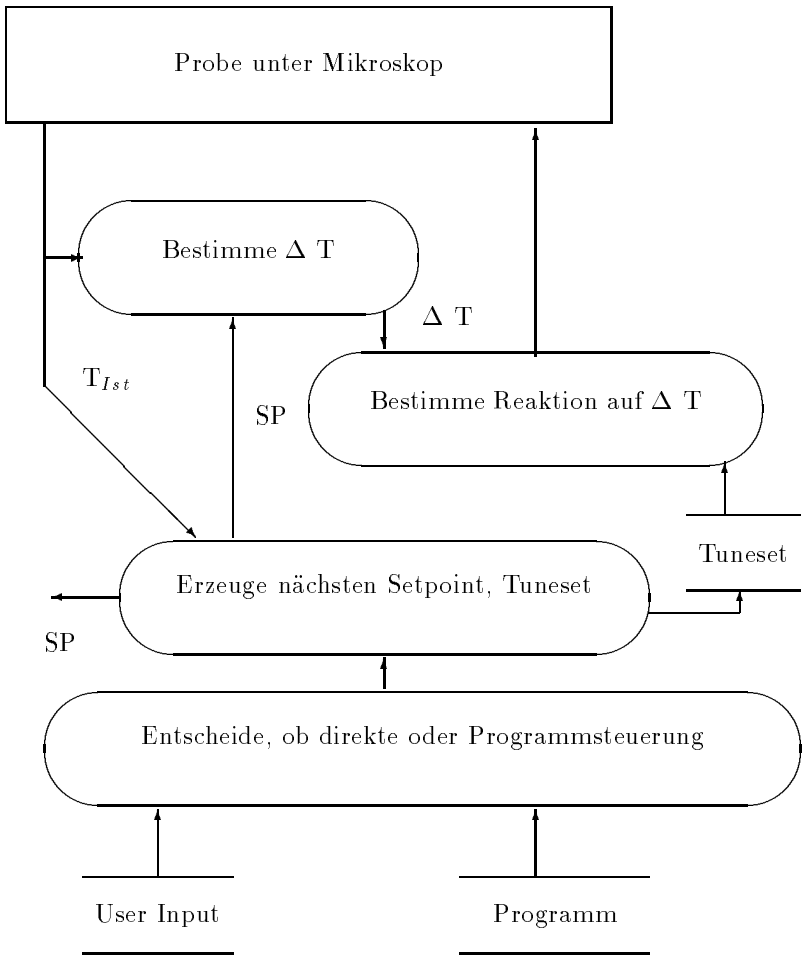

Abb 4 Temperaturverlauf erzeugen

Abb. 4 stellt den Vorgang *Temperaturverlauf erzeugen* näher dar. Dabei werden *Bestimme* ΔT und *Bestimme Reaktion auf* ΔT durch den Omega Controller erledigt.

Name	Produziert von	Verwendet von	Zweck/Beschreibung	Form
PID			Werte zur Steuerung des Regelkreises für Temperaturkontrolle	Reals mit einer Nachkommastelle
Temperatur				Zahl mit einer Nachkommastelle aus dem Bereich -500,0 ... +1000,0
Aussehen der Probe	Mikroskop	Bestimme Veränderung	Feststellen von Veränderungen	Bild
TIst (Isttemperatur)	Mikroskop (Heiz/ Kühltisch)	• Kalibriere Temp • Bestimme SP-TIst • Erzeuge nächsten Setpoint, Tuneset • Erstelle Statusinfo	Die Temperatur, die augenblicklich am Heiz-/Kühltisch des Mikroskops anliegt.	Temperatur

Tab 1 Legende zu den Abbildungen

Name	Produziert von	Verwendet von	Zweck/Beschreibung	Form
Setpoint (=SP)	Erzeuge nächsten Setpoint, Tuneset	• Bestimme SP-TIst • Erstelle Statusinfo	Welche Temperatur der Heiz/Kühltisch annehmen soll. Optional mit Zeit und Toleranzangabe.	• Temperatur • Anzahl Sekunden • Toleranz: Zahl ≥ 0 mit einer Nachkommastelle.
Veränderung	Bestimme Veränderung	Erzeuge Ereignis	Art der Veränderung, um auf deren Ursache schließen zu können	Text
TKal	Kalibriere Temp	• Erzeuge Ereignis • Erstelle Statusinfo	Ist-Temp gemäß Temperaturdrift etc. kalibriert.	Temperatur
ΔT	Bestimme SP-TIst	Bestimme Reaktion auf ΔT	Bestimmen der Abweichung Ist $\leftrightarrow$ Soll	

Tab 2 Legende zu den Abbildungen (Fortsetzung)

Name	Produziert von	Verwendet von	Zweck/Beschreibung	Form
Ereignis	Erzeuge Ereignis	Ereignisliste	Protokolliert, welche Temperatur zu welcher Veränderung führt	• Temperatur • Text
StatusInfo	Erstelle StatusInfo	Statusbericht	Um Experimentfortschritt zu erkennen.	• Soll: Temperatur • Ist: Temperatur • Zeit: Anzahl Sekunden seit Experimentbeginn (ohne Pausen)
Tuneset	Erzeuge nächsten Setpoint, Tuneset	Bestimme Reaktion auf ΔT	Beeinflußt die Reaktion auf Temperaturabweichungen.	• Heizen:PID • Kühlen:PID

Tab 3 Legende zu den Abbildungen (Fortsetzung)

2.2.3 Spezifikation der System Requirements

2.2.3.1 Usercharakteristika

Der typische User ist ein Diplomand, Dissertant oder ein Student aus dem Bereich der Geologie und Paläontologie. Daraus ergibt sich auch der Einsatzbereich des Systems: Es dient einerseits zur Forschung, andererseits auch zur Ausbildung im Rahmen von Übungen bzw. Labors.

2.2.3.2 Behavioral Requirements

MS-Windows 3.1 hält Zeitvorgaben nicht immer genau ein. Diese Tatsache ist bei den folgenden Zeitangaben berücksichtigt. Sofern nicht besonders angegeben, sind alle Temperaturangaben in Celsius darzustellen.

Der Anregung von Gene Callahan in [Cal94] folgend, werden keine Toolbars verwendet. Die Verwendung von Icons und ähnlichen graphische Spielereien sind auf das notwendige Mindestmaß zu reduzieren. Nach Möglichkeit werden die Windows Standarddialoge in der Borland Fassung (BWCC) verwendet.

Das Computersystem hat folgende Aufgaben zu erfüllen:

1. Akustische Rückmeldungen generieren
2. Überwachen des Omega Controllers
3. Während eines Experiments den Omega Controller steuern
4. Den Temperaturverlauf während eines Experiments darstellen
5. Informationen über den Controller darstellen
6. Ereignisse protokollieren
7. Ereignisse darstellen
8. Das Ereignisprotokoll bearbeiten lassen
9. Erstellen und Auswählen von Tunesets
10. Kalibrierung der Temperaturwerte

Rückmeldungen durch das System

Es gibt folgende Feedbackereignisse:

TCS-Warning Versuch, das Programm zu beenden, wenn das Experiment noch nicht gespeichert wurde.

TCS-TempReached Die gewünschte Temperatur liegt am Heiz/Kühltisch an.

TCS-Done Allgemeines "Fertig"-Signal. Außerdem verwendet, um das Löschen eines Ereignisses zu signalisieren.

TCS-Alert Signalisiert Alarmzustände wie eine Unterbrechung der Verbindung zum Omega Controller, einen Resourcenmangel von Windows, eine Überschreitung der Sicherheitstemperatur und ähnliches.

TCS-CantDoThat Die gewünschte Funktion kann nicht ausgeführt werden (z.B. falsches Paßwort oder ähnliches).

TCS-Working Mitteilung, daß ein Profile aktiv ist oder daß das Programm ein zweites Mal gestartet wurde.

TCS-Acknowledged Entweder wurde vom Benutzer eine neue Temperatur spezifiziert oder ein Ereignis wurde generiert.

Für alle diese Ereignisse kann in der Datei WIN.INI unter dem Abschnitt [Sound] eine WAV-Datei angegeben werden. Wird keine angegeben, so ist dieses Feedbackereignis deaktiviert. Sofern die WAV-Datei nicht existiert oder keine WAV-Fileausgabe möglich ist, wird ein Standardsignal über den PC-Lautsprecher erzeugt.

Überwachen des Omega Controllers

Es gibt folgende Sicherheitsparameter:

Sicherheitsparameter	**Bedeutung**
TMax	Maximaler erlaubter Temperaturwert.
TLo, THi	Untere bzw. obere Grenze des sicheren Temperaturbereiches, d.h. jenes Bereiches, in dem der Heiz/Kühltisch gefahrlos angefaßt werden kann.
fSound	Flag zur Warntonausgabe
fBreakable	Flag für Abbruch der Sicherheitsabschaltung

Tab 4 Sicherheitsparameter

Die Temperatur des Omega Controllers wird in Abständen von 1 Sekunde abgefragt und mit dem Wert TMax verglichen. Wird dieser Wert überschritten, wird die Sicherheitsabschaltung ausgelöst.

Bei der Sicherheitsabschaltung wird als neuer Temperatur-Sollwert $(TLo + THi)/2$ eingestellt. Während der Sicherheitsabschaltung wird das Experiment nicht abgebrochen. Sobald die Temperatur des Heiz/Kühltisches den Sollwert erreicht oder sogar unterschritten hat, wird die Sicherheitsabschaltung beendet. Damit endet auch das Experiment.

Wenn fSound gesetzt ist, ertönt während der Sicherheitsabschaltung in Abständen von 20 Sekunden ein Alarmsignal. Dieses Alarmsignal wird durch die Angabe einer WAV-Datei in WIN.INI in der Sektion [Sounds] unter dem Schlüsselwort TCS-Alert bestimmt.

In Abhängigkeit des Wertes fBreakable kann die Sicherheitsabschaltung abgebrochen werden. Befindet sich nach einem Abbruch die Temperatur des Heiz/-Kühltisches aber noch jenseits des Wertes TMax, wird wieder eine Sicherheitsabschaltung ausgelöst.

Der Mechanismus der Sicherheitsabschaltung kann auch vom User direkt ausgelöst werden. Dann ist die Sicherheitsabschaltung immer abbrechbar und es ertönt kein Warnsignal. Die vom User ausgelöste Sicherheitsabschaltung wird

beendet, sobald die Temperatur des Heiz/Kühltisches einen Wert innerhalb des Bereiches $[TLo, THi]$ angenommen hat.

Während eines Experiments den Omega Controller zu steuern

Es gibt zwei Arten, wie der User ein Experiment durchführen kann:

1. profile-gesteuert
2. durch direkte Eingaben

Profile-Modus

Um die Erzeugung von Temperaturverläufen möglichst flexibel zu gestalten, wird dafür eine eigene Programmiersprache verwendet. Diese Sprache wird ab Seite 36 definiert. Ein gültiges Programm in dieser Sprache wird Profile genannt.

Der User gibt einen Dateinamen einer ANSI-Datei an, die den Source-Text eines Profiles enthält. Dieses Profile wird vom Computer auf syntaktische Fehler überprüft. Wenn ein solcher gefunden wird, wird das Profile nicht ausgeführt. Bei syntaktischer Korrektheit wird das Experiment gestartet (bzw. fortgesetzt). Tritt ein Laufzeitfehler auf, so wird das Experiment abgebrochen und eine Fehlermeldung angezeigt. Die bis dahin gesammelten Experimentdaten werden nicht gelöscht.

Der Benutzer kann zu jedem Zeitpunkt das Profile anhalten bzw. abbrechen, und in den Direkt-Eingabe-Modus wechseln. Wird das Profile nur angehalten, kann nach Verlassen des Direkt-Eingabe-Modus das Profile fortgesetzt werden. Ein abgebrochenes Profile kann nur neu gestartet werden.

Nach Beenden des Profiles wird das Experiment automatisch angehalten.

Direkt-Eingabe-Modus

Es erscheint ein Dialogfenster, in dem der User Soll-Temperaturen, Toleranzen und Zeiten angeben kann. Weiters kann die Temperatureinheit (K..Kelvin, C..Celsius, F..Fahrenheit) eingestellt werden. Der Direkt-Eingabe-Modus muß durch den User verlassen werden, um das Experiment anzuhalten. Wenn der Controller die Temperatur erreicht hat, wird ein akustisches Signal ausgegeben.

Darstellung des Temperaturverlaufs

Im oberen Bereich des Hauptfensters wird der Temperaturverlauf wie folgt dargestellt:

- Der Hintergrund dieses Bereichs ist weiß.
- Die Zeit wird horizontal aufgetragen. Ihr Startwert und ihr Bereich werden durch 2 horizontale Scrollbars eingestellt, die sich unterhalb der Graphik befinden. Die Zeitskala wird auf Höhe der 0 Grad Temperaturlinie dargestellt. An ihrem linken bzw. rechten Rand wird die dargestellte Zeit angezeigt. Sollte die 0 Grad Linie unter oder über dem dargestellten Temperaturbereich liegen, so werden die Zeiten am unteren bzw. oberen Rand der Graphik angezeigt.

- Die Temperatur wird vertikal aufgetragen. Der Startwert und der Bereich werden durch 2 vertikale Scrollbars eingestellt, die sich links neben der Graphik befinden. Am linken Rand der Graphik befindet sich die Temperaturskala. An deren oberem und unterem Bildrand wird die entsprechende Temperatur angezeigt.
- Der Verlauf der Soll-Temperatur wird als grünes Polygon dargestellt.
- Der Verlauf der Ist-Temperatur wird als rotes Polygon dargestellt.

Informationen über Omega Controller darstellen

Am unteren linken Rand des Hauptfensters wird eine gut sichtbare Temperaturanzeige dargestellt. Zusätzlich zur Temperaturanzeige werden drei Statusfelder mit folgender Bedeutung von links nach rechts dargestellt:

Modus zeigt den Betriebsmodus des Omega Controllers an.

Toleranz signalisiert die Verwendung eines Toleranzwertes für die Temperatur.

Warten zeigt an, ob eine Kommunikation mit dem Omega Controller stattfindet.

Der *Betriebsmodus* wird durch folgende Farbcodierung dargestellt.

1. rot: Der Omega Controller regelt die Temperatur nicht.
2. gelb: Der Omega Controller regelt die Temperatur nur mit einem Soll-Wert.
3. blau: Der Omega Controller regelt die Temperatur mit einer Zeit- und/-oder einer Toleranzvorgabe.

Wird eine *Toleranz* verwendet, so ist die Anzeige "Toleranz" grün, ansonsten grau.

Wenn das System mit dem Omega Controller kommuniziert, ist die Anzeige "Warten" rot, ansonsten grün. Wäehrend "Warten" rot ist, reagiert das TCS auf keine Benutzereingaben.

Wenn die Verbindung zum Controller unterbrochen ist, wird als Temperatur -999.9 angezeigt.

Ereignisse protokollieren

Das Erzeugen bzw. Löschen eines Ereignisses ist nur möglich, wenn ein Experiment aktiv ist.

Der User kann mit einem Eingabegerät ein Ereignis erzeugen oder löschen. Ereignisse werden in einer LIFO-Reihenfolge gespeichert bzw. gelöscht. Der Versuch, bei einer leeren Ereignisqueue ein Ereignis zu löschen, hat keine Auswirkungen. Das Erzeugen und Löschen von Ereignissen wird bei vorhandener Hardware durch unterschiedliche Soundereignisse begleitet. Das Schlüsselwort für das Erzeugen eines Ereignisses ist TCS-Acknowledged, für das Löschen eines Ereignisses wird TCS-Done verwendet.

Ereignisse anzeigen

Die Ereignisse werden graphisch im Hauptfenster angezeigt. Dafür wird ein Bereich verwendet, der genau so hoch ist wie das Fenster zur Darstellung der Statusinformation des Omega Controllers und sich vom rechten Rand der Statusinformation bis zum rechten Rand des Hauptfensters erstreckt. Am rechten Rand dieses Bereiches befindet sich ein Scrollbar gleicher Höhe, der zum Einstellen des Temperaturbereiches dient. Am unteren Rand befindet sich ein horizontaler Scrollbar, mit dem die Starttemperatur des Bereiches am linken Rand eingestellt wird.

In der linken bzw. rechten oberen Hälfte wird die Temperatur des linken bzw. rechten Bereichsendes dargestellt. In der Mitte dieses Bereichs wird eine horizontale Skalierung eingeblendet. Der Wert 0 Grad Celsius wird als strichlierte weiße senkrechte Linie, die über den ganzen vertikalen Bereich geht, dargestellt. Ereignisse, die eine negative Temperatur besitzen, werden durch blaue senkrechte durchgezogene Linien dargestellt. Ereignisse, deren Temperatur positiv ist, werden durch gelbe senkrechte durchgezogene Linien dargestellt. Ein Ereignis mit der Temperatur von 0 Grad Celsius wird mit einer weißen senkrechten durchgezogenen Linie dargestellt.

Wenn kein Experiment aktiv ist, so wird bei einem Click mit der linken Maustaste innerhalb dieses Bereiches gesucht, welche Ereignisse in der Nähe der Mausposition sind. Als Nähe gilt die Toleranzbreite, die Windows als Doppelclick akzeptiert, multipliziert mit dem ΔT, welches sich aus der Anzeige ergibt:

$$\Delta T = Temperaturbereich/Breite\ der\ Darstellung\ in\ Pixel.$$

Alle gefundenen Ereignisse werden in einem Fenster ohne Rand und Titelzeile dargestellt. Die Darstellung der Ereignisse beinhaltet die Temperatur und eine Beschreibung des Ereignisses, sofern eine solche angegeben wurde. Wenn kein Ereignis gefunden wird, so wird "Keine Einträge gefunden" angezeigt. Das Fenster ist so groß zu wählen, daß der anzuzeigende Text darin Platz findet, dennoch soll es so klein wie möglich sein. Ein Mausclick mit der rechten Maustaste oder ein Mausclick mit der linken Maustaste außerhalb dieses Bereiches bringt dieses Fenster zum Verschwinden. Ebenso verschwindet das Fenster, wenn das Experiment fortgesetzt wird.

Bearbeiten der Ereignisliste

Mittels eines eigenen Menüpunktes kann die Liste der Ereignisse bearbeitet werden. Die Ereignisse werden dazu nach Temperatur sortiert und in einem Fenster angezeigt. Ereignisse können gelöscht werden und ihre Bezeichnung kann geändert werden. Neue Ereignisse können nicht erzeugt werden, ebenso können gelöschte Ereignisse nicht regeneriert werden. Weiters kann die Ereignisliste als reine ANSI-Datei gespeichert werden. Nach Beenden der Bearbeitung wird für die verbleibenden Ereignisse die ursprüngliche LIFO-Ordnung wiederhergestellt.

Erstellen und Auswählen von Tunesets

Es wird zwischen 2 Modi unterschieden:

1. Erstellen von Tunesets
2. Auswählen von Tunesets

Beide Modi verwenden den selben Dialog, im Modus "Auswählen" sind aber sowohl ein Ändern der angezeigten Daten sowie das Speichern deaktiviert. Der Modus "Erstellen von Tunesets" ist durch ein Paßwort zu schützen. Das Paßwort ist case-sensitiv und darf neben äöüÄÖÜß alle Zeichen enthalten, die einen ANSI-Code besitzen, der kleiner als 127 ist.

Beim Öffnen dieses Dialogs wird das aktuelle Tuneset des Omega Controllers gelesen und angezeigt. Es stehen folgende Befehle zur Verfügung:

Fertig Der Dialog wird geschlossen. Eventuelle ungesicherte Änderungen werden verworfen.

Senden Das angezeigte Tuneset wird zum Controller übertragen. Im Falle des Dialogs zum Erstellen des Tunesets liegt es im Verantwortungsbereich des Benutzers, gültige und sinnvolle Werte einzutragen.

Lesen Das aktuelle Tuneset des Omega Controllers wird gelesen.

Laden Es erscheint ein Dialogfeld, in dem ein Tunesetfile gewählt werden kann. Dieses wird geladen und angezeigt. Um es auch zum Omega Controller zu übertragen, muß *Senden* gewählt werden.

Speichern Es erscheint ein Dialogfeld, in dem der Name für das Tuneset eingegeben werden kann. Dieser Button ist im Modus "Auswählen von Tunesets" grau und kann nicht gedrückt werden.

Für das Laden und Speichern werden die Standarddialoge von Windows verwendet.

Kalibrierung

Die Kalibrierungstabelle ist eine ANSI/ASCII-Textdatei. Ihr Format besteht aus Fließkommazahlen gemäß der Profilesyntax, die durch Whitespace getrennt sind. Ebenso wie bei Profilen sind Kommentare innerhalb { und } erlaubt. Andere Zeichen stellen einen Fehler dar. Die Zahlen werden der Reihe nach eingelesen und als Zahlenpaare (TMess, TKal) interpretiert, wobei TMess dem gemessenen Temperaturwert und TKal dem "wahren" Temperaturwert entspricht. Gemessene Werte, die sich nicht in der Tabelle befinden, werden mit einem interpolierten Wert kalibriert. Vergleiche dazu die formale Spezifikation der Kalibrierungstabelle auf Seite 32.

3. FORMALE SPEZIFIKATIONEN

Als erstes stellt sich folgende Frage: "Warum eine zweite Spezifikation, wenn es diese bereits in Abschnitt 2.2.3 auf Seite 23 gibt?" Tatsächlich ist Abschnitt 2.2.3, den ich in der Folge als *informelle Spezifikation* bezeichnen möchte, auch für den Laien verständlich und beschreibt recht genau, welche Aufgaben zu erfüllen sind.

Der Schlüssel zur Beantwortung der obigen Frage liegt in "für den Laien verständlich" und "beschreibt recht genau". Daß auch ein Laie die informelle Spezifikation versteht, wäre an und für sich kein Nachteil, bedeutet aber, daß eine normale Sprache verwendet wird. Natürliche Sprachen zeichnen sich aber dadurch aus, daß sie in vielen Bereichen ungenau sind. Darüber hinaus sind für den Laien andere Punkte wichtig als für den Programmierer. Als drastisches Beispiel betrachte man den Abschnitt über die Profilesprache auf Seite 25: In diesem kurzen Abschnitt wird alles gesagt, was der Kunde wissen muß, um beurteilen zu können, ob die Aufgabe adäquat erfüllt wird. Der Programmierer fängt mit dieser Beschreibung allein aber nichts an.

Jeanette Wing in [Win90] beschreibt sehr gut, welche Eigenschaften eine Spezifikationssprache haben muß, um als solche zu gelten. Sie stellt dar, welche Gruppe (User, Programmierer, Auftraggeber, usw.) welche Vorteile aus Spezifikationen zieht und belegt dies anhand von Beispielen. Nico Plat [PvKT92] versucht die Gründe darzulegen, warum der Wert formaler Spezifikationen in der Praxis nicht recht erkannt wird. Er zeigt an einem konkreten Beispiel, wie eine formale Spezifikation verwendet wird. Zwei der Schwierigkeiten, die dem Einsatz von formalen Spezifikationen entgegenstehen sind:

1. Die beteiligten Personen sind aufgrund ihrer Ausbildung meist nicht in der Lage, mit formalen Spezifikationen umzugehen.

2. Die praktischen Vorteile werden nicht erkannt, man sieht nur den Aufwand und nicht den Nutzen.

Der Nutzen aber ist vielfältig und als Beispiel seien hier nur angeführt: Durch eine formale Spezifikation werden Ungenauigkeiten und Mehrdeutigkeiten der natürlichen Sprache beseitigt und dies ermöglicht eine (zumindest teilweise automatisiert ablaufende) Beweisbarkeit auf Richtigkeit der Spezifikation und somit des fertigen Produktes.

Die Motivation für die formale Spezifikation ist es also, dem Programmierer genau zu sagen, *was* er zu implementieren hat. Die Herausforderung dabei ist, gleichzeitig *nicht* zu sagen, *wie* er zu implementieren hat (außer in gut begründeten Ausnahmen). Da die formale Spezifikation ein Leitfaden für den Programmierer ist, welcher der Implementation nicht vorwegnehmen soll, werden die Requirements nur sehr abstrakt spezifiziert. Für Typen werden daher

nur Bezeichner eingeführt, ihre Realisierung interessiert nicht weiter. Funktionen werden *ihrer Wirkung nach* beschrieben. Der Grund dafür ist die Absicht, die Spezifikation unabhängig bezüglich Betriebssystem, Programmiersprache, Hardware und ähnlichem zu halten. Im Idealfall wird zuerst die Spezifikation erledigt und danach die Auswahl der zu verwendenden Komponenten (Hardware, Betriebssystem, Bibliotheken...) getroffen. In der Realität ist dies nicht immer möglich, da bestimmte Entscheidungen bereits getroffen sind (z.B. Computertyp und Betriebssystem). Im konkretem Fall war praktisch alles bereits vorhanden. Trotzdem ist es sinnvoll, die Spezifikation auch in einem solchem Fall wie im Idealfall (d.h. möglichst abstrakt) durchzuführen, da eine eventuelle Portierung erleichtert wird.

Da die formale Spezifikation beschreibt, was zu implementieren ist, ist sie auch ein Teil der Dokumentation. Auch hier ist es wieder von Vorteil, daß Implementationsdetails nicht erfaßt werden, da der Überblick nicht durch eine Fülle von Details verloren geht, die für die Beschreibung der Aufgaben nicht notwendig ist.

Die folgenden Spezifikationen verwenden die RAISE - Sprache, wie sie in [Geo95] beschrieben wird.

3.1 Ereignis

Ein Ereignis beinhaltet eine Temperatur und eine Beschreibung. Ein Ereignis wird mit *create* erzeugt, wobei ihm eine Beschreibung und eine Temperatur zugewiesen werden. Die Temperatur kann mit *temp* abgefragt werden, *desc* (description) liefert die Beschreibung. Warum für Temperatur und Beschreibung je eine Funktion deklariert und nicht ein Tupel verwendet wird, wird in Abschnitt 5.1 auf Seite 69 ersichtlich.

```
TEREIGNIS =
class
  type TEreignis,TTemperatur,TDescription


  value
    create : TTemperatur × TDescription → TEreignis
    temp : TEreignis → TTemperatur,
    desc : TEreignis → TDescription
end
```

Diese Ereignisse werden in einer Ereignisliste gesammelt. Es sind mehrere gleiche Ereignisse in dieser Liste gestattet. Elemente werden immer am Ende der Liste hinzugefügt und können in umgekehrter Reihenfolge wieder entfernt werden (undo). Weiters können Ereignisse auch in beliebiger Reihenfolge entfernt werden, wobei dann allerdings bei mehreren gleichen Ereignissen pro Entfernungsvorgang nur jeweils ein Ereignis, welches implementationsabhängig ausgewählt wird, entfernt wird. Die Beschreibung von einzelnen Ereignissen kann geändert werden, jedoch wird immer nur bei maximal einem Ereignis die Beschreibung geändert.

add fügt ein Ereignis in die Ereignisliste ein.

undo löscht ein Ereignis in umgekehrter Reihenfolge des Einfügens.

remove löscht ein Ereignis aus der Liste.

change ändert die Beschreibung eines Ereignisses.

```
TEREIGNISLIST =
class
  type TEreignis,TDescription
       TEreignisList = TEreignis*


  value
    empty : TEreignisList,
    add : TEreignisList × TEreignis → TEreignisList,
    undo : TEreignisList → TEreignisList,
    remove : TEreignisList × TEreignis → TEreignisList,
    change : TEreignisList × TDescription ×
             TDescription → TEreignisList,


  axiom ∀ e,e1:TEreignis, l:TEreignisList, d1,d2:TDescription •
    undo(empty) ≡ empty,
    undo(add(l,e)) ≡ l,


    add(empty,e) ≡ [e],


    remove(empty,e) ≡  empty,
    remove(add(l,e),e1) ≡
      if e1 = e then l
                else add(remove(l,e1),e),


    change(empty,e) ≡ empty,
    change(add(l,e),d1,d2) ≡
      let t=temp(e),
          d=desc(e) in
        if d=d1 then add(l,create(t,d2))
                     else add(change(l,d1,d2),e)
  end
```

3.2 Experiment

Ein Experiment besteht aus einer Experimentstatusliste und einer Ereignisliste. Als Experimentstatus wird die Soll- und die Isttemperatur zu einem bestimmten Zeitpunkt bezeichnet. Während eines Experiments werden die Experimentstati kontinuierlich gesammelt. Ein Experiment ist durch eine Experimentstatusliste und eine Ereignisliste festgelegt.

```
type TExperimentStatus = ( TTemperatur, TTemperatur, TTime)
```

```
type TExperimentStatusList = TExperimentStatus*
```

```
type TExperiment = (TExperimentStatusList, TEreignisList)
```

3.3 Temperatur

Definition des Typs der Temperaturangaben:
Temperaturangaben besitzen genau eine Nachkommastelle, und besitzen daher folgende Definition.

```
type TTemperatur = R
     inv t:TTemperatur == t ∈ {x | (x*10) ∈ Z ∧ (−1000,0 < x < 1000,0)}
```

3.4 Kalibrierungstabelle

Die Kalibrierungstabelle besteht aus Paaren (t_i,t_s), wobei t_i den gemessenen und t_s den tatsächlichen Temperaturwert bezeichnet. Bei der Kalibrierung der Temperatur t gibt es drei Fälle:

1. $t = t_i$. Dann wird der dazugehörende Wert t_s zurückgegeben.
2. $t_{i1} < t < t_{i2}$. Die T-Kurve zwischen (t_{i1},t_{s1}) und (t_{i2},t_{s2}) wird als Gerade interpoliert und t_s wird dementsprechend berechnet. (Siehe die Funktion *Eval*).
3. $t < min(t_i)$ oder $t > max(t_i)$. Das heißt, der Temperaturwert t liegt nicht in dem von der Tabelle abgedeckten Bereich. Dann wird die Temperaturkurve gemäß ihrer Steigung an ihrem passenden Rand als Gerade extrapoliert und t_s dementsprechend berechnet.

Der Index in die Kalibrierungstabelle KalTab beginnt bei 0.

```
KALIBTAB =
class
  type KalEntry = (TTemperatur,TTemperatur),
       KalTab = KalEntry*,
       TTemperatur


  value
    empty : KalTab,
    add : KalTab × KalEntry → KalTab,
    kalib : KalTab × TTemperatur → TTemperatur,
    Eval  : KalEntry × KalEntry → (float × float),


  axioms ∀ t_i,t_s,t'_i,t'_s:TTemperatur, k:KalTab, e:KalEntry •
    add(empty,e) ≡ [e],


    add(add(k,(t_i,t_s)),e) ≡
      let (t'_i,t'_s)=e
      in if t_i < t'_i then add(add(k,e),(t_i,t_s))
                       else [e] ^ add(k,(t_i,t_s))


    kalib(empty,t) ≡  t
    kalib(k,t) ≡
      let el=findmaxLoEq(k,t),
          eh=findminHiEq(k,t),
          (l_i,l_s) = el,
          (h_i,h_s) = eh
      in if el=eh then l_s
         else if el=nil then let (k,d)=Eval(eh,k[1])
              else if eh=nil then let (k,d)=Eval(k[len k - 2],el)
                   else let (k,d)=Eval(el,eh)
             in t*k+d


    Eval(a,b) = let (a_i,a_s)=a,
                    (b_i,b_s)=b,
                    k =(b_s-a_s) / (b_i-a_i)
                in (k,(a_s-k*a_i))
end
```

KalEntry ist ein Eintrag in der Tabelle, bestehend aus gemessener Temperatur t_i und "wahrer" Temperatur t_s.

add trägt einen *KalEntry* nach t_i aufsteigend sortiert in der Kalibrierungstabelle ein.

kalib führt die Kalibrierung durch. Wird der zu kalibrierende Temperaturwert in der Tabelle nicht gefunden, so wird inter- bzw. extrapoliert.

Eval führt die Inter- bzw. Extrapolation durch.

findmaxLoEq findet das Paar (t_i, t_s), sodaß $t_i \leq t$ und t_i maximal ist.

findminHiEq findet das Paar (t_i, t_s), sodaß $t_i \geq t$ und t_i minimal ist.

3.5 Sicherheitsparameter

Jene Parameter, die das Verhalten der Sicherheitsabschaltung beeinflussen, werden in diesem Typ TSicherheitsparameter zusammengefaßt.

```
type TSicherheitsparameter = (TMax, (TLo,THi),fBreakable,fSound)
```

TMax : TTemperatur ist jener Temperaturwert, der maximal erlaubt ist. Überschreitet die Ist-Temperatur diesen Wert, wird eine Sicherheitsabschaltung eingeleitet.

(TLo,THi) : TTemperatur × *TTemperatur* gibt einen Temperaturbereich (genannt Sicherheitsbereich) an, in welchem der Heiz-/Kühltisch gefahrlos berührt werden kann. Im Rahmen einer Sicherheitsabschaltung oder auf direktem Kommando wird die Soll-Temperatur auf

$$(TLo + THi)/2$$

gesetzt. Die manuelle Sicherheitsabschaltung wird beendet, sobald die Ist-Temperatur innerhalb dieses Intervalls liegt, die automatische bei Erreichen oder Unterschreiten des Sollwerts.

fBreakable : BOOLEAN Wenn fBreakable=True, dann kann die Sicherheitsabschaltung abgebrochen werden, bevor die Ist-Temperatur des Heiz-/Kühltisches innerhalb des Sicherheitsbereichs ist.

fSound : BOOLEAN Wenn fSound=True, dann wird während der Sicherheitsabschaltung ein Warnton in 20-Sekundenintervallen erzeugt.

3.6 Tuneset

Ein Tuneset besteht aus zwei PID-Tripeln und wird zur Steuerung der Temperaturregelung verwendet. Ein PID-Wert bestimmt die Regelung zur Kühlung und der andere beeinflußt den Heizvorgang. Für eine Beschreibung der PID-Werte siehe Seite 79.

$$\texttt{type PID} = ((x, y, z) : (Integer \times Real \times Real) \mid x \in \{5..4000\} \wedge (y * 100) : Integer \in \{0..2000\} \wedge (z * 100) : Integer \in \{0..2000\})$$

TUNESET = (Heiz,Kuehl:PID)

4. SPEZIFIKATION DER PROFILSPRACHE

4.1 Profile

Die zu erfüllende Aufgabe ist das programmgesteuerte Erzeugen von Temperaturwerten, die an den Omega Controller weitergeleitet werden. Es gibt zwei Methoden, mit denen diese Aufgabe erfüllt werden kann: Die Verwendung eines API (Application Programming Interface), das heißt einer Sammlung von Unterprogrammen, oder einer eigenen Sprache.

Ein API hätte folgende Vorteile:

+ Es wären nur 5 Funktionen (Temperatur setzen, Temperatur lesen, Tuneset lesen, Tuneset schreiben, Start/Stop des Omega Controllers) und einige Hilfsfunktionen wie Initialisieren des Controllers zu spezifizieren bzw. zu implementieren.

+ Die Implementation als VxD (Virtueller Gerätetreiber von Windows) oder als DLL (Dynamic Link Library) hätte einen zusätzlichen Anreiz geboten, Geräteunabhängigkeit zu realisieren.

+ Jede beliebige Programmiersprache, mit der ein Windowsprogramm erzeugt werden kann, hätte zur Erzeugung der Temperaturwerte herangezogen werden können.

Die Nachteile sind aber groß:

- Es muß ein Windowsprogramm erstellt werden, und dies ist kein trivialer Vorgang. Unter Windows 95 (oder Windows NT) hätte man auf eine Console-Applikation (d.h. nur Textmodus) zurückgreifen können. Diese steht aber unter Windows 3.11 (dem Zielsystem) nicht in passabler Form zur Verfügung.

- Der Benutzer muß mit 2 Programmen gleichzeitig arbeiten (das TCS und das Programm zur Erzeugung der Temperaturwerte).

- Die Erzeugung der Temperaturwerte kann nicht durch das TCS angehalten und später fortgesetzt werden.

Auf der anderen Seite besitzt eine eigene Sprache, die vom TCS ausgeführt wird, folgende Vorteile:

+ Es ist kein Windowsprogramm zu erstellen, der Benutzer kann sich (fast) vollständig auf die eigentliche Aufgabe (Erstellen des Temperaturverlaufs) konzentrieren.

+ Es wird nur ein Programm verwendet, in welchem die Temperatursteuerung integriert ist.

+ Man kann Makros zum Erstellen von häufigen Temperaturverläufen verwenden.

Der Vollständigkeit halber auch die Nachteile einer eigenen Sprache:

- Eine neue (wenn auch einfache) Sprache ist zu lernen.

- Für 5 Funktionen muß ein relativ großer Aufwand betrieben werden (Vergleiche dazu 4.5).

Aufgrund des Gesagten (oder besser Geschriebenen) entschied ich mich für die eigene Sprache, da sie für den Benutzer, bei dem keine weitreichenden Informatikkenntnisse vorausgesetzt werden dürfen (Siehe 2.2.3.1 auf Seite 23), einfacher ist. Zusätzlich ermöglicht diese Entscheidung, die Spezifikation einer Programmiersprache zu untersuchen.

Um eine Programmiersprache zu spezifizieren, muß man 3 Dinge getrennt behandeln ([Wat91]):

1. Syntax
2. Semantik
3. Laufzeitverhalten

Obwohl jeder der 3 Punkte wichtig ist, wird es von 1 bis 3 immer schwieriger, genau und vollständig zu sein.

4.2 Konkrete Syntax eines Profiles

Am einfachsten ist die Spezifikation der konkreten Syntax. In ihr wird festgelegt, wie der Sourcecode aussieht, welche Trennzeichen verwendet werden, wie die Befehle tatsächlich heißen und wie sie aufgebaut sind und ob Teile eines Befehls optional sind. Sie bestimmt, ob zwischen Groß- und Kleinschreibung unterschieden wird und vieles mehr. Um die Syntax festzulegen, wird die Backus Naur Form (BNF) verwendet ([Ten81]).

4.2.1 Programmstruktur

```
<profile> ::== <profilebody>
              |<profiledecllist> <profilebody>
```

Dieses ist die erste und wichtigste Regel, weil `<profile>` in keiner anderen Regel auftaucht. Durch das sukzessive Ersetzen der Regelvariablen auf der rechten Seite durch passende Varianten bis nur mehr Terminalsymbole vorhanden sind, kann jedes gültige Profile beschrieben werden. Als Terminalsymbole gelten zum Beispiel `PROFILE` und `ENDPROFILE` aus der nächsten Regel, aber auch Operatoren etc. Obwohl es sehr viele Erweiterungen der BNF gibt, die speziellen Zeichen eine spezielle Bedeutung geben, halte ich mich an eine Basis-BNF, d.h. die einzigen speziellen Symbole sind ::== und —. Alle anderen (Sonder-)Zeichen, sofern sie nicht Teil einer Regelvariable sind, sind Teil der spezifizierten Syntax.

```
<profilebody> ::== PROFILE <stmtlist> ENDPROFILE

<profiledecl> ::== <fundecl>
                 | <subdecl>
                 | <varstmt>
                 | <includestmt>

<profiledecllist> ::== <profiledecl>
                     |<profiledecl> <profiledecllist>

<stmtlist> ::== <generalstmt>
              |<generalstmt> <stmtlist>
```

4.2.2 Statements

```
<generalstmt> ::== <varstmt>
                  |<tsreadstmt>
                  |<tswritestmt>
                  |<tspushstmt>
                  |<tspopstmt>
                  |<rampstmt>
                  |<beepstmt>
                  |<waitstmt>
                  |<ifstmt>
                  |<whilestmt>
                  |<labelstmt>
                  |<gotostmt>
                  |<callstmt>
                  |<assignstmt>
                  |<printstmt>
                  |<inputstmt>
                  |<openstmt>
                  |<closestmt>
                  |<fileprintstmt>
                  |<fileinputstmt>
```

```
                  |<deletestmt>
                  |<chdirstmt>
                  |<includestmt>

<assignstmt> ::== <variable> := <expr>;

<beepstmt> ::== BEEP;
               |BEEP <strexpr>;
               |BEEP <strexpr>,<intexpr>;

<callstmt> ::== CALL <identifier>;
               |CALL <identifier> (<paramlist>);
<chdirstmt> ::== CHDIR <strexpr>;

<closestmt> ::== CLOSE #<intexpr>;
<deletestmt> ::== DELETE <strexpr>;
<fileinputstmt> ::== INPUT #<intexpr>, <variable>;
<fileprintstmt> ::== PRINT #<intexpr>, <exprlist>;

<inputstmt> ::== INPUT ?<strexpr>, <variable>;
                |INPUT <variable>;

<openstmt> ::== OPEN <strexpr> AS #<intexpr> FOR <fopenmode>;
<printstmt> ::== PRINT <expr>;
                |PRINT <expr>, <exprlist>;
<rampstmt> ::== RAMP <realexpr>;
               |RAMP <realexpr> IN <intexpr>;
               |RAMP <realexpr> IN <intexpr> WITHIN <realexpr>;
               |RAMP <realexpr> WITHIN <realexpr>;

<tspopstmt> ::== TSPOP;

<tspushstmt> ::== TSPUSH;
<tsreadstmt> ::== TSREAD <strexpr>;
<tswritestmt>::== TSWRITE<strexpr>;
<waitstmt> ::== WAIT;
               |WAIT <strexpr>;
               |WAIT <strexpr>,<strexpr>;
               |WAIT <strexpr>,<strexpr>,<intexpr>;
```

4.2.3 Kontrollstrukturen

Oberflächlich betrachtet sind die folgenden Kommandos auch Statements. Da sie aber den Fluß der Abarbeitung der Anweisungen signifikant verändern (im Gegensatz zu Funktionen und Unterprogrammen) verdienen sie eine eigene Überschrift.

```
<gotostmt> ::== GOTO <identifier>;
<labelstmt> ::== : <identifier>

<ifstmt> ::== IF <numberexpr> THEN <stmtlist> ENDIF;
             |IF <numberexpr> THEN <stmtlist>
                              ELSE <stmtlist> ENDIF;

<whilestmt> ::== WHILE <numberexpr> DO <stmtlist> WEND;
```

4.2.4 Deklaration von Funktionen und Unterprogrammen

Eine Funktion ist Teil eines Ausdrucks und kann einen Wert zurückliefen. Daher benötigt sie einen Typ und ein `Return`-Statement, daß einen Ausdruck benötigt. Ein Unterprogramm liefert keinen Wert zurück und kann nicht Teil eines Ausdrucks sein. Daher verwendet es nur ein einfaches `Return`-Statement. Ansonsten unterscheiden sich Funktions- und Unterprogrammdeklaration nicht.

```
<fundecl> ::== FUN <identifier> : <typ>;
                   <funbody>
                ENDFUN;
               |FUN <identifier> (<paramdecllist>):<typ>;
                   <funbody>
                ENDFUN;
<funbody> ::== <funstmt>
              | <funstmt> <funbody>
<funreturnstmt> ::== RETURN <expr>;
<funstmt> ::== <funreturnstmt> | <generalstmt>

<subdecl>::== SUB <identifier>; <subbody> ENDSUB;
             |SUB <identifier> (<paramdecllist>);
                 <subbody>
              ENDSUB;
<subbody> ::== <substmt>
              |<substmt> <subbody>
<subreturnstmt> ::== RETURN;
<substmt> ::== <subreturnstmt> | <generalstmt>
```

4.2.5 Parameter

```
<paramdecllist> ::== <paramdecl>
                    |<paramdecl>, <paramdecllist>

<paramdecl>::== <ref-paramdecl> | <val-paramdecl>
```

```
<parameter> ::== <val-parameter>
                |<ref-parameter>

<paramlist> ::== <parameter>
                |<parameter>, <paramlist>
<paramname> ::== <identifier>
                |<identifier> []

<paramnamelist> ::== <paramname>
                  |<paramname>, <paramnamelist>

<ref-paramdecl> ::== VAR <val-paramdecl>

<ref-parameter> ::== <variable>

<val-paramdecl> ::== <paramnamelist> : <typ>

<val-parameter> ::== <expr>
```

4.2.6 Ausdrücke

```
<expr> ::== <strexpr>
           |<numberexpr>

<intexpr> ::== <intexpr> <intop> <intexpr>
              |<intkonst>
              |<variable>
              |<funktion>
              |<expr> <compop> <expr>

<numberexpr> ::== <realexpr>
                 |<intexpr>

<realexpr>::== <realexpr> <realop> <realexpr>
              |<intexpr>
              |<realkonst>
              |<variable>
              |<funktion>

<strexpr> ::== <strexpr> <strop> <strexpr>
              |<strkonst>
              |<variable>
              |<funktion>

<strkonstlist> ::== <strkonst>
                   | <strkonst>, <strkonstlist>
```

4.2.7 Operatoren

```
<compop> ::== < | <= | = | >= | > | <>
<intop> ::== + | - | * | / | MOD | <compop>
<realop> ::== + | - | * | /
<strop> ::== +
```

4.2.8 Variablen

```
<varstmt> ::== VAR <vardecllist>;

<arrayname> ::== <identifier> [<intkonst>]
<identifier> ::== <identifierhead>
                 |<identifierhead> <identifiertail>

<identifierhead> ::== <letter> | _
<identifiertail> ::== <identifiertailbody>
                     |<identifiertailbody> <identifiertail>

<identifiertailbody>::== <identifierhead>
                         |<digit>

<simplename>::== <identifier>

<typ> ::== REAL | INTEGER | STRING
<vardecl> ::== <varnamelist> : <typ>
<vardecllist> ::== <vardecl>
                  |<vardecl>, <vardecllist>
<variable> ::= <identifier>
              |<identifier>[<intexpr>]

<varname> ::== <arrayname>
              |<simplename>

<varnamelist> ::== <varname>
                  |<varname>,<varnamelist>
```

4.2.9 Konstanten

```
<konst> ::== <realkonst>
            |<intkonst>
            |<strkonst>

<expkonst> ::= <sign> <stdintkonst>
              |<stdintkonst>

<intkonst> ::== <sign> <uintkonst>
               |<uintkonst>

<realkonst> ::== <sign> <urealkonst>
                |<urealkonst>

<scirealkonst> ::== <stdrealkonst> E <expkonst>

<stdintkonst> ::== <digit>
                  |<digit> <stdintkonst>

<stdrealkonst> ::== <stdintkonst>
                   |<stdintkonst>.<stdintkonst>

<strkonst> :: == <strdelimiter1> <char>* <strdelimiter1>
               |<strdelimiter2> <char>* <strdelimiter2>
<timeintkonst> ::== <stdintkonst>
                   |<stdintkonst> : <stdintkonst>
                   |<stdintkonst> : <stdintkonst> : <stdintkonst>

<uintkonst> ::== <stdintkonst> | <timeintkonst>

<urealkonst> ::== <stdrealkonst>
                 |<scirealkonst>
```

4.2.10 Sonstiges

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<exprlist> ::== <expr>
               |<expr>, <exprlist>

<fopenmode> ::== INPUT
                |OUTPUT
                |APPEND
```

```
<letter> ::= A | B | ... | Y | Z | a | b | ... | y | z |
<sign> ::== + | -

<strdelimiter1> ::== ¨
<strdelimiter2> ::== ‘

<includestmt> ::== INCLUDE <strkonstlist>;
```

Das `<includestmt>` ist eine Compileranweisung und deshalb kein eigentliches Statement.

Als `<char>` gilt jedes Zeichen des Computerzeichensatzes.

Kommentare beginnen mit { und enden mit }. Sie dürfen überall dort vorkommen, wo auch ein Leerzeichen vorkommen darf und können beliebig verschachtelt werden. Alles (auch Compileranweisungen) innerhalb eines Kommentars wird ignoriert.

4.3 Semantik eines Profiles

Auf den ersten Blick könnten wir jetzt aufhören und uns mit der syntaktischen Korrektheit zufrieden geben. Bereits auf den zweiten Blick wird aber sichtbar, daß syntaktische Korrektheit nicht gleichzeitig auch "korrektes Programm" bedeutet. So sind die Anweisungen

```
A := 1/0;
S := “17“;
ramp S within A;
```

syntaktisch vollkommen korrekt. Für eine semantische Korrektheit sind aber zusätzliche Dinge zu überprüfen, z.B:

- Sind A und S bereits definierte Variablen?
- Stimmen die Typen der Variablen?
- Ist die Division durch 0 überhaupt erlaubt?
- Gibt es eine implizite Typkonversion (z.B. Integer → Real, String → Real)?
- Stimmen die Typen für den ramp Befehl?

Die Antworten auf diese Fragen gibt nur die Semantik und definiert daher, wie die Sprache zu interpretieren ist (Siehe auch [JS90]).

Die Semantik besteht aus 2 Teilen:

1. statische Semantik
2. dynamische Semantik

Ad 1.) Die statische Semantik besteht aus einer Vielzahl von Definitionen und Regeln, die einen Algorithmus in die Lage versetzen, beurteilen zu können, ob ein Ausdruck in einer Sprache *gültig* ist, das heißt, ob die Elemente des Ausdrucks in ihrem Zusammenhang (= Kontext) sinnvoll sind (Deshalb heißen diese Regeln auch *Kontextbedingungen*). Das bekannteste Beispiel eines solchen Algorithmusses ist ein Compiler, der aber neben der Semantik auch gleich die Syntax mitüberprüft. Da ein Algorithmus diese Gültigkeitsbeurteilung durchführt ist sofort die Grenze der Möglichkeiten klar. Laufzeitverhalten kann aufgrund des Halteproblems (siehe [HR81]) aber nicht von einem Algorithmus überprüft werden.

Ad 2.) Die dynamische Semantik beginnt dort, wo die statische endet und betrachtet das Laufzeitverhalten. Hier wird definiert, wie ein Ausdruck umgesetzt wird. Die dynamische Semantik kann die Korrektheit aber nur sehr rudimentär überprüfen und stellt eigentlich nur Regeln auf, wie Sprachelemente zu interpretieren sind.

4.3.1 Abstrakte Syntax

Wie schon vorhin erwähnt, reicht die konkrete Syntax nicht aus, um eine Sprache zu beschreiben. Ein wesentlicher Punkt sind auch die Typen von Ausdrücken und Parametern. Auf den ersten Blick scheint die konkrete Syntax diese Information zu liefern, schließlich gibt es ja zum Beispiel die Definition einer `<intkonst>` und was kann das schon anderes sein als eine Integerkonstante? Obwohl diese Beobachtung im konkreten Fall richtig ist, gilt sie allgemein nicht, da es z.B. möglich ist, alle Definitionen in der BNF anders zu benennen, z.B. `<a>,<aa>,<ab>...` und solange die Austauschfunktion bijektiv ist, ändert sich nichts an der Korrektheit und der tatsächlichen Aussagekraft der BNF. Was verloren geht, ist, neben der Lesbarkeit, eine informelle Vereinbarung, eine Integerkonstante auch gefälligst so zu bezeichnen, daß jeder sofort weiß, daß eine Integerkonstante gemeint ist.

Andererseits sind viele Informationen aus der BNF für die Semantik unwichtig. So ist es völlig egal, mit welchem Zeichen Parameter getrennt werden, wie eine Variablendeklaration aussieht, mit welchem Schlüsselwort Call-By-Reference-Parameter gekennzeichnet werden (wenn überhaupt) und so weiter.

Daher definiert man nun die *Abstrakte Syntax*, welche sich genau auf diese Informationen beschränkt, in dem sie Typen verwendet. Zur Darstellung wird RSL verwendet und es wird jedem Konstrukt aus der konkreten Syntax ein Ausdruck in der abstrakten Syntax zugeordnet. Sie ist daher auch sehr kompakt, aber allein wenig aussagekräftig. Deshalb benötigt man auch noch die *Kontextbedingungen* (Siehe Seite 47). Dieser Abschnitt folgt der Beschreibung in [BJ78].

```
Profile         :: dl:Decl* Main:Block

ArrayAssign     :: d:ArrayVar s:ArrayVar
ArrayVar        :: vm:ID
ArrayVarDekl    :: a-vn:ID a-size:Const
ArrayType       =  Integer* | Real* | String*
AssignStmt      =  SimpleAssign | ArrayAssign
```

```
BeepStmt        :: sname:Expr sIntervall:Expr
Block           :: sl:Statement*
CallType        =  CallByReference | CallByValue | Local
Char            =  Implementation defined set
ChdirStmt       :: n:Expr
CloseStmt       :: fnr:Expr
Const           :: Real | Integer | String
Decl            =  SubDecl | FunDecl | VarStmt
DeleteStmt      :: n:Expr
Expr            =  InfixExpr | PraefixExpr |
                   SimpleVar | SubscrVar |
                   Const | FuncCall
FuncCall        :: fn:ID pl:Parameter*
FunDecl         :: fname:ID pl:Parameter* ftp:SimpleType
                   fbody:Block*
FunReturnStmt   :: re:Expr
GotoStmt        :: l:ID
ID              :: Infinite Set
IfStmt          :: b:Expr b-then:Block b-else:Block
InfixExpr       :: e1:Expr op:InfixOp e2:Expr
InfixOp         =  REAL-PLUS | REAL-MINUS |
                   REAL-MUL | REAL-DIV |
                   INT-PLUS | INT-MINUS |
                   INT-MUL | INT-DIV | INT-MOD |
                   STRING-PLUS |
                   LT | LE | EQ | GE | GT | NE
InputFStmt      :: fnr:Expr d:SimpleLValue
InputStmt       :: p:Expr d:SimpleLValue
Integer         =  Implementation defined set (embedded in Real)
LabelDecl       :: l:ID
LValue          =  SimpleLValue | ArrayVar
NotifyStmt      =  BeepStmt | WaitStmt
OpenMode        =  "Open" | "Input" | "Append"
OpenStmt        :: n:Expr fnr:Expr mode:OpenMode
Param-bRef      =  LValue
Param-bVal      :: bv:Expr
Parameter       =  Param-bVal | Param-bRef
PraefixExpr     :: op:PraefixOp e:Expr
PraefixOp       =  REAL-PLUS | REAL-MINUS | INT-PLUS | INT-MINUS
PrintFStmt      :: fnr:Expr el:Expr+
PrintStmt       :: el:Expr+
RampStmt        :: temp:Expr Time:Expr Tol:Expr
Real            =  Implementation defined set
SimpleAssign    :: d:SimpleLValue e:Expr
SimpleLValue    =  SimpleVar | SubscrVar
SimpleType      =  INTEGER | REAL | STRING
SimpleVar       :: vn:ID
SimpleVarDekl   :: s-vn:ID
Statement       =  VarStmt | TunesetStmt | TempStmt |
                   NotifyStmt |
```

```
                  IfStmt | WhileStmt |
                  GotoStmt | ReturnStmt |
                  SubCall | AssignStmt |
                  PrintStmt | InputStmt |
                  OpenStmt | CloseStmt |
                  PrintFStmt | InputFStmt |
                  DeleteStmt | ChdirStmt |
                  LabelDecl
String          = Char*
SubCall         :: sn:ID pl:Parameter*
SubDecl         :: sname:ID pl:Parameter* sbody:Block
SubscrVar       :: vn:ID index:IntExpr
TSReadStmt      :: tsname:Expr
TSWriteStmt     :: tsname:Expr
TunesetStmt     = TSReadStmt | TSWriteStmt |
                  TSPushStmt | TSPopStmt
Type            = SimpleType | ArrayType
VarDekl         :: vl:VarnameDekl+ tp:SimpleType
VarnameDekl     = SimpleVarDekl | ArrayVarDekl
VarStmt         = VarDekl+
WaitStmt        :: Message:Expr sname:Expr sIntervall:Expr
WhileStmt       :: b:Expr body:Block
```

4.3.2 Kontext Bedingung

Die Kontextbedingungen bestehen aus einer Reihe von `is-wf- ...` (is-wellformed- ...) Regeln, welche für jedes Element der Sprache definieren, wie ein "wohlgeformtes" Element aussieht.

Im wesentlichen wird dabei darauf geachtet, daß Typen von Variablen, Parametern und Ausdrücken kompatibel sind, daß Variablen, Funktionen und Unterprogramme vor ihrer ersten Verwendung definiert werden und ähnliches.

Für diese Bedingungen wird nun eine "Umgebung" (engl. *Environment*) verwendet. In diesem Environment wird jeder Identifier mit seinem Typ festgehalten. In der Folge bezeichnet die Variable env immer das aktuell gültige Environment.

Dieser Abschnitt folgt der Beschreibung in [BJ78] unter der Überschrift "Context Conditions". Es wird nur eine statische Typprüfung vorgenommen und Bereichsüberschreitungen lösen Laufzeitfehler während des Programmablaufs aus.

4.3.2.1 Environment

Das Environment besteht aus folgenden, von einander getrennten Abschnitten:

FunSub enthält die Namen und Parameterlisten von Unterprogrammen und Funktionen.

Profil enthält die Variablen des Hauptprogramms. Jede Variablendeklaration, die nicht innerhalb einer Deklaration eines Unterprogramms oder einer Funktion ist, gilt nur im Hauptprogramm.

Lokal enthält die Variablen und Parameter von einem Unterprogramm bzw. einer Funktion. Weiters sind Labeldeklarationen von Unterprogrammen bzw. Funktionen enthalten.

```
Environment = [Profil → Env, Lokal → Env, FunSub → FSEnv]

Env = [Label → LabelEnv, Var → IdEnv]

LabelEnv = [name → defined]
defined = boolean

IdEnv = [name → (typ × kind × mode)]
typ = {INTEGER, REAL, STRING, void}
kind = {simple,array}
mode = {var, byVal, byRef}

FunSub = [name → (typ × paramlist)]
```

Diese entwas komplizierte und durchaus unübliche Einteilung des Environments ergibt sich aus folgenden Eigenschaften der Sprache:

1. Funktionen, Unterprogramme und das Hauptprogramm sind voneinander total getrennt. Der einzige Weg, Werte in eine Funktion bzw. ein Unterprogramm zu bekommen, ist über Parameter. Werte aus einer Funktion bzw. Unterprogramm kann man nur erhalten über den Funktionswert (natürlich nur bei Funktionen) und über Call-By-Reference Parameter.

2. Es gibt keine globalen Variablen. Wenn eine Variable nicht innerhalb einer Funktion oder eines Unterprogramms definiert wird, gehört sie zum Hauptprogramm. Da aber Variablen des Hauptprogramms außer in Deklarationen von Funktionen bzw. Unterprogrammen überall definiert werden können, ist die Trennung in ein Environment *Profil* und ein Environment *lokal* notwendig.

3. Es gibt keine verschachtelten Funktions- bzw. Unterprogrammdeklarationen. Daher ist es nicht notwendig, eine Liste (oder einen Stapel) von lokalen Environments zu definieren, man kommt mit einem lokalen Environment aus.

4. Funktionsnamen und Unterprogrammnamen sind global gültig, deshalb stehen sie in einem eigenen Environment. Sollte eine Funktion gleich wie eine Variable benannt werden, hat die Variablendefinition Vorrang. Bei Unterprogrammaufrufen ist aufgrund des Call-Befehls klar, daß es sich um ein Unterprogramm handelt.

5. Da Sprünge nur innerhalb eines Unterprogramms bzw. einer Funktion bzw. des Hauptprogramms stattfinden dürfen, ist es nur logisch, die Label-Definitionen in das passende Environment (Profil oder Lokal) zu geben.

6. Da Sprungziele auf schon verwendet werden können, bevor sie definiert werden, ist es notwendig, im Environment zu unterscheiden, ob ein Label bereits definiert wurde oder nicht.

4.3.2.2 Zuweisungen

```
is-wf-AssignStmt(a,env) read ENVIRONMENT =
  is-wf-SimpleAssign(a,env) ∨ is-wf-ArrayAssign(a,env)

is-wf-ArrayAssign(mk-ArrayAssign(Destination,Source),env)
read ENVIRONMENT =
  is-type-array(Destination) &
  is-type-array(Source) &
  TypesAreCompatible(TypeOf(Source,env),TypeOf(Destination,env))&
  MaxIndex(Source,env)=MaxIndex(Destination,env) &
  Destination ∈ dom ENVIRONMENT(env)(Var) &
  Source ∈ dom ENVIRONMENT(env)(Var)

is-wf-SimpleAssign(mk-SimpleAssign(Destination,Expr),env)
read ENVIRONMENT=
  is-type-simple(Destination,env) &
  is-wf-Expr(Expr,env) &
  TypesAreCompatible(TypeOf(Expr,env), TypeOf(Destination,env))&
  Destination ∈ dom ENVIRONMENT(env)(Var)
```

4.3.2.3 Variablendeklaration

```
is-wf-VarDekl(mk-VarDekl(vl,tp),env) write ENVIRONMENT=
  tp ∈ {INTEGER, REAL, STRING} &
  ( len vl > 0 &
    ∀ i ∈{1 ... len vl } →
    let v = vl[i] in
      is-wf-SimpleVarDekl(v,tp,env)
      ∨ is-wf-ArrayVarDekl(v,tp,env)
  )

is-wf-ArrayVarDekl(mk-ArrayVarDekl(vn,vrange),tp,env)
write ENVIRONMENT =
  not (vn ∈ dom ENVIRONMENT(env)(Var)) &
  typeof(vrange)=INTEGER &
  let ENVIRONMENT(env)(Var)=ENVIRONMENT(env)(Var)
                            † [vn → (tp,array,var)]
  in true
```

```
is-wf-SimpleVarDekl(mk-SimpleVarDekl(v),tp,env)
write ENVIRONMENT =
  not (v ∈ dom ENVIRONMENT(env)) &
  let ENVIRONMENT(env)(Var)=ENVIRONMENT(env)(Var)
                           † [v → (tp,simple,var)]
  in true
```

4.3.2.4 Ausdrücke

```
is-wf-Expr(e,env) read ENVIRONMENT =
  is-wf-InfixExpr(e,env) ∨ is-wf-PraefixExpr(e,env) ∨
  is-wf-SimpleVar(e,env) ∨ is-wf-SubscrVar(e,env)  ∨
  is-wf-Const(e,env) ∨ is-wf-FuncCall(e,env)

is-wf-InfixExpr(mk-InfixExpr(e1,op,e2),env) read ENVIRONMENT =
  is-wf-Expr(e1,env) & is-wf-Expr(e2) &
  let te1=TypeOf(e1,env),
      te2=TypeOf(e2,env)
  in cases op:
       REAL-PLUS, REAL-MINUS,
       REAL-MUL, REAL-DIV → TypesAreCompatible(te1,te2) &
                            (te1=REAL ∨ te2=REAL)
       INT-PLUS, INT-MINUS,
       INT-MUL, INT-DIV, INT-MOD →
                           te1=INTEGER & te2=INTEGER
       STRING-PLUS →        te1=STRING & te2=STRING
       LT | LE | EQ |
       GE | GT | NE →       TypesAreCompatible(te1,te2)

is-wf-PraefixExpr(mk-PraefixExpr(op,e),env) read ENVIRONMENT =
  is-wf-Expr(e,env) &
  let te = TypeOf(e,env)
  in cases op:
       REAL-PLUS, REAL-MINUS → te=REAL
       INT-PLUS, INT-MINUS   → te=INTEGER

is-wf-SimpleVar(mk-SimpleVar(name),env) read ENVIRONMENT =
  name ∈ dom ENVIRONMENT(env)(Var) &
  is-type-simple(name)

is-wf-SubscrVar(mk-SubscrVar(name,Index),env) read ENVIRONMENT =
  is-type-array(name,env) &
  is-wf-Expr(Index,env) &
  TypeOf(Index,env)=INTEGER &
  name ∈ dom ENVIRONMENT(env)(Var)
```

4.3.2.5 Spezielle Befehle für TCS

```
is-wf-BeepStmt(mk-BeepStmt(sname,sIntervall),env)
read ENVIRONMENT =
  is-wf-Expr(sname,env) &
  is-wf-Expr(sIntervallenv) &
  TypeOf(sname,env)=STRING &
  TypeOf(sIntervall,env)=INTEGER

is-wf-RampStmt(mk-RampStmt(Temp,Time,Tol),env) read ENVIRONMENT =
  is-wf-Expr(Temp,env) &
  is-wf-Expr(Time,env) &
  is-wf-Expr(Tol,env) &
  (TypeOf(Temp,env)=REAL ∨ TypeOf(Temp,env)=INTEGER) &
  TypeOf(Time,env)=INTEGER &
  (TypeOf(Tol,env)=REAL ∨ TypeOf(Tol,env)=INTEGER)

is-wf-TSReadStmt(mk-TSReadStmt(tsname),env) read ENVIRONMENT =
  is-wf-Expr(tsname,env) &
  TypeOf(tsname,env)=STRING

is-wf-TSWriteStmt(mk-TSWriteStmt(tsname),env) read ENVIRONMENT =
  is-wf-Expr(tsname,env) &
  TypeOf(tsname,env)=STRING

is-wf-WaitStmt(mk-WaitStmt(Message,sname,sIntervall),env) =
  is-wf-Expr(Message,env) &
  is-wf-Expr(sname,env) &
  is-wf-Expr(sIntervall,env) &
  TypeOf(Message,env)=STRING &
  TypeOf(sname,env)=STRING &
  TypeOf(sIntervall,env)=INTEGER
```

4.3.2.6 Wrapperbefehle

Die folgenden Befehle ermöglichen den Zugriff auf Windowsfunktionen zur Ein/Ausgabe. Mit Ausnahme von InputStmt und PrintStmt beziehen sich alle Befehle auf Dateioperationen, die praktisch unverändert an das Betriebssystem weitergeleitet werden. Open, InputF, PrintF, und Close verwenden eine Dateinummer wie unter Basic üblich, um das Konzept des Dateihandles vor dem Benutzer zu verbergen.

```
is-wf-ChdirStmt(mk-ChdirStmt(Name),env) read ENVIRONMENT =
  is-wf-Expr(Name,env) &
  TypeOf(Name,env)=STRING

is-wf-CloseStmt(mk-CloseStmt(fnr),env) read ENVIRONMENT =
  is-wf-Expr(fnr,env) &
  TypeOf(fnr,env)=INTEGER

is-wf-DeleteStmt(mk-DeleteStmt(Name),env) read ENVIRONMENT =
  is-wf-Expr(Name,env) &
  TypeOf(Name,env)=STRING

is-wf-InputFStmt(mk-InputFStmt(fnr,Destination),env)
read ENVIRONMENT =
  TypeOf(fnr,env)=INTEGER &
  is-type-simple(Destination,env) &
  TypeOf(Destination,env)=STRING &
  Destination ∈ dom ENVIRONMENT(env)(Var)

is-wf-InputStmt(mk-InputStmt(Prompt,Destination),env)
read ENVIRONMENT =
  TypeOf(Prompt)=STRING &
  is-type-simple(Destination,env) &
  Destination ∈ dom ENVIRONMENT(env)(Var)

is-wf-OpenStmt(mk-OpenStmt(name,fnr,mode),env)
read ENVIRONMENT =
  is-wf-Expr(name,env) &
  is-wf-Expr(fnr,env) &
  TypeOf(name,env)=STRING &
  TypeOf(fnr,env)=INTEGER &
  mode ∈ {<APPEND>,<INPUT>,<OUTPUT>}

is-wf-PrintFStmt(mk-PrintFStmt(fnr,el),env) read ENVIRONMENT =
  is-wf-Expr(fnr,env) &
  TypeOf(fnr,env)=INTEGER &
  len el > 0 &
  ∀ e ∈ el → is-wf-Expr(e,env)

is-wf-PrintStmt(mk-PrintStmt(el),env) read ENVIRONMENT =
  len el > 0 &
  ∀ e ∈ el → is-wf-Expr(e,env)
```

4.3.2.7 Kontrollstrukturen

```
is-wf-GotoStmt(mk-GotoStmt(l),env) write ENVIRONMENT =
   let if not l ∈ ENVIRONMENT(env)(Label) then
       ENVIRONMENT(env)(Label)=ENVIRONMENT(env)(Label)
                                † [l → true]
   in true

is-wf-LabelDecl(mk-LabelDecl(l),env) write ENVIRONMENT =
  if l ∈ dom ENVIRONMENT(env)(Label) then
    let (name,defined) = ENVIRONMENT(env)(Label)(l)
    in if defined then false
       else let ENVIRONMENT(env)(Label)=ENVIRONMENT(env)(Label)
                                         † [l → true]
            in true
  else let ENVIRONMENT(env)(Label)=ENVIRONMENT(env)(Label)
                                    † [l → true]
       in true
```

Ein Label darf in einem Block nur einmal definiert werden. Bei einer Vorwärtsreferenz eines Goto-Befehls wird das Label vorläufig akzeptiert, muß aber spätestens am Ende des Blocks definiert sein. Sonst liegt ein Fehler vor.

```
is-wf-IfStmt(mk-IfStmt(b,b-then,b-else),env) read ENVIRONMENT =
  is-wf-Expr(b,env) &
  TypeOf(b,env)=INTEGER &
  is-wf-Block(b-then) &
  is-wf-Block(b-else)

is-wf-WhileStmt(mk-WhileStmt(b,body),env) read ENVIRONMENT =
  is-wf-Expr(b,env) &
  TypeOf(b,env)=INTEGER &
  is-wf-Block(body,env)
```

4.3.2.8 Funktionen und Unterprogramme

```
is-wf-ActParamList(fname,aktParamlist,env) write ENVIRONMENT =
  let defParamList=GetParamList(fname,env)
    len defParamList = len aktParamList &
    (∀ i ∈ {1..len defParamList} →
       let defParam=defParamList(i),
           aktParam=aktParamList(i)
         (cases GetCallType(defParam):
            CallByValue →
              TypeOf(defParam,env)=TypeOf(aktParam,env) &
```

```
                is-wf-Expr(aktParam,env)
             CallByReference →
                TypeOf(defParam,env)=TypeOf(aktParam,env)
                &  (
                      (is-type-simple(defParam,env) &
                       is-type-simple(aktParam,env)
                      ) ∨ (is-type-array(defParam,env) &
                                is-type-array(aktParam,env)
                               )
                   )
          )
    )

is-wf-FunBody(mk-FunBody(sl,ftyp),env) write ENVIRONMENT =
  len sl = 0 ∨
  let fs=hd sl
    in (if fs = mk-FunReturn(e) then is-wf-FunReturn(fs,ftyp,env)
                               else is-wf-Statement(fs),env)
       ∧ is-wf-FunBody(mk-FunBody((tl sl,ftyp)),env)

is-wf-FuncCall(mk-FuncCall(fname,aktParamList),env)
read ENVIRONMENT =
  is-type-func(fname,env) &
  is-wf-ActParamList(fname,aktParamList,env) &
  fname ∈ dom ENVIRONMENT(env)(FunSub)

is-wf-FunDecl(mk-FunDecl(fname,defParamList,ftyp,fbody),env)
write ENVIRONMENT =
  not fname ∈ dom ENVIRONMENT(FunSub) &
  ftyp ∈ {STRING, INTEGER, REAL } &
  ( len defParamList = 0 ∨ is-wf-defParamList(defParamList) )
  & let ENVIRONMENT(Lokal) = mk-NewEnv(defParamList),
        ENVIRONMENT(FunSub) = ENVIRONMENT(FunSub)
                             † [fname → (ftyp × defParamList)]
     in is-wf-FunBody((fbody,ftyp),Lokal)

is-wf-FunReturn(mk-FunReturn(e),funtype,env) read ENVIRONMENT =
  is-wf-Expr(e) & (TypesAreCompatible(TypeOf(e,env),funtype))

is-wf-SubBody(mk-SubBody(sl),env) write ENVIRONMENT =
  len sl = 0 ∨
   let fs=hd sl
   in (fs = mk-SubReturn() ∨ is-wf-Statement(fs)) &
       is-wf-SubBody(mk-SubBody(tl sl),env)

is-wf-SubCall(mk-SubCall(fname,aktParamList),env)
read ENVIRONMENT =
  is-type-proc(fname,env) &
  is-wf-ActParamList(fname,aktParamList,env) &
```

```
  fname ∈ dom ENVIRONMENT(env)(FunSub)

is-wf-SubDecl(mk-SubDecl(fname,defParamList,fbody))
write ENVIRONMENT =
  not fname ∈ dom ENVIRONMENT(FunSub) &
  ( len defParamList = 0 ∨
    is-wf-defParamList(defParamList) )
  & let ENVIRONMENT(Lokal) = mk-NewEnv(defParamList)
        ENVIRONMENT(FunSub) = ENVIRONMENT(FunSub)
                              † [fname → (void × defParamList)]
    in is-wf-SubBody(fbody,Lokal)
```

4.3.2.9 Sonstiges

```
is-wf-Block(sl,env) write ENVIRONMENT =
  (len sl = 0 ∨ (is-wf-Statement(hd sl))
                ∧ is-wf-Block(tl sl,env)) &
  not false ∈ rng(ENVIRONMENT(env)(Labels))


is-wf-Profile(mk-Profile(dl,Main)) write ENVIRONMENT =
  (dl=[] ∨ ∀ i ∈ {1 ... len dl} →
                              (is-wf-SubDecl(dl(i),Profil) ∨
                               is-wf-FunDecl(dl(i),Profil) ∨
                               is-wf-VarDekl(dl(i),Profil)
                              )
  ) & is-wf-Block(Main,Profil)
```

4.3.2.10 Hilfsfunktionen

```
TypesAreCompatible(t1,t2) = (t1=t2)
                          ∨ (t1=Integer ∧ t2=REAL)
                          ∨ (t1=REAL ∧ t1=Integer)
```

`mk-NewEnv(defParamList)` erstellt aufgrund der Parameterliste ein neues Environment für eine Funktion oder ein Unterprogramm.

`is-wf-const(x,env)` liefert TRUE, wenn x eine Konstante ist.

`is-wf-Statement(s,env)` überprüft, ob s im Environment env ein gültiges Statement ist, in dem die passende is-wf- Funktion aufgerufen wird.

`is-type-array(x,env)` liefert TRUE, wenn x ein Array ist.

`is-type-simple(x,env)` liefert TRUE, wenn x eine einfache Variable oder ein Element eines Arrays ist.

`is-type-func(fname,env)` ist TRUE, wenn fname eine Funktion bezeichnet.

`is-type-proc(fname,env)` ist TRUE, wenn fname ein Unterprogramm (SUB) bezeichnet.

`MaxIndex(array,env)` liefert das obere Ende des Indexbereichs von array.

`TypeOf(x,env)` liefert den Typ von x.

`GetParamList(fname,env)` liefert die Parameterliste für die Funktion/Subroutine fname.

4.4 Laufzeitverhalten eines Profiles

Die Spezifikation des Laufzeitverhaltens ist für general-purpose-Sprachen sehr umfangreich. Die Nützlichkeit und die Effizienz werden dadurch sehr stark beeinflußt. Bei special-purpose-Sprachen kann man sich aber auf die wesentlichsten Dinge beschränken.

Als Beispiel dient hierfür wieder meine Programmiersprache, die im wesentlichen 2 Aufgaben erfüllt:

1. Erlaubt eine programmgesteuerte Kontrolle des Omega Controllers.
2. Gestattet das Schreiben eines Programms unter MS-Windows, ohne daß der Benutzer ein MS-Windowsprogramm schreiben muß.

Der zweite Punkt ist deshalb wichtig, weil die Erstellung eines Windows-Programms auch trotz Tools wie Visual Basic relativ komplex ist, ohne daß der Verwender des TCS die Fähigkeiten, die mit dieser Komplexität einhergehen, benötigt. Daher sind auch die meisten Befehle einfache Wrapper für Windowsfunktionen.

Da die Spezifikation des Laufzeitverhaltens der Implementation nicht vorgreifen soll (siehe dazu den Beginn dieses Kapitels), beschränkt sie sich auf eine phänomenologische Beschreibung, die in der Regel trotzdem sehr umfangreich ist. Da hier aber eine special-purpose-Sprache vorliegt, erlaube ich es mir, mich auf die Beschreibung der einzelnen Befehle und Kontrollstrukturen zu beschränken. In Anlehnung an [NN92] führe ich das Laufzeitverhalten als operationelle Semantik an, wobei ich sie auf einen Pseudocode abbilde.

In der nachfolgenden Spezifikation wird entgegen der Richtlinie der abstrakten Spezifikation einige Male konkret auf den Omega Controller eingegangen (z.B. bei der Spezifikation des `ramp`-Befehls). Eine hardwarwe-unabhängige Spezifikation hätte bedeutet, in der gesammten Spezifikation der Requirements eine geräteunabhängige Methode zur Temperaturveränderung zu finden, nur um bei der Implementation erst wieder auf das Omega Controller Modell zurückgreifen zu müssen. Das heißt aber auch, daß ein sehr großer Aufwand für das Finden eines unabhängigen Modells hätte verwendet werden müssen, ohne daß ein konkreter Nutzen argumentierbar gewesen wäre (siehe dazu auch Seite 75).

4.4.1 Sprache für die Spezifikation

Die Spezifikationen in diesem Abschnitt besitzen folgende Form:

Pre *preconditions*

Inv *invariants*

Post *postconditions*

[fun] *Sprachelement*
code

Sprachelement gibt an, welcher Befehl oder welche Funktion spezifiziert wird. Allfällige Parameter werden in Klammern angeführt. Funktionen werden mit *fun* eingeleitet. Ihr Funktionsergebnis wird nur durch den *return*-Befehl bestimmt.

Preconditions sind Zustände und Bedingungen, die vor der Ausführung von *code* gültig sein müssen. Sind sie es nicht, erfolgt eine Laufzeitfehlermeldung oder das Verhalten von *code* ist nicht definiert (implementationsabhängig).

Invariants sind *wichtige* Bedingungen und Zustände, die sich während der Ausführung von *code* nicht ändern. In der Regel wird hier auf Dinge eingegangen, von denen es nicht offensichtlich ist, daß sie nicht verändert werden.

Postconditions sind Bedingungen und Zustände, die nach Ausführung von *code* gültig sind.

code ist die Beschreibung des Laufzeitverhaltens des Sprachelements. Dafür verwende ich einen Pseudocode, der Elemente von SML und Pascal vermischt und um mathematische Konstrukte wie $x \in \{\}$ erweitert wurde.

Die Precondistions, Invariants und Postconditions werden nur informell beschrieben.

4.4.2 Programmablauf

Ein Profile ist eine Liste von Befehlen. Die Befehle sind der Reihe nach eindeutig durchnummeriert. Weiters gibt es ein Environment, in dem die aktuellen Variablen mit ihren Werten gespeichert sind. Bei Programmstart wird die Liste der Befehle, ein Startindex und ein Hauptenvironment an die Executeprozedur übergeben. Unterprogramm- bzw. Funktionsaufrufe werden durch einen rekursiven Aufruf dieser Executeprozedur realisiert.

Befehle oder Funktionen, die mit OS... beginnen, beschreiben Routinen, die vom Betriebssystem zur Verfügung gestellt werden.

4.4.2.1 MakeNewEnvironment(envDescription,paramlist)

Die Aufgabe von MakeNewEnvironment ist es, gemäß der Beschreibung in envDescription ein neues Environment zu erzeugen und mit den Werten aus paramlist zu initialisieren. Das Environment selbst ist eine Liste von Variablen dar. Bei der Erzeugung eines Environments werden 3 Fälle unterschieden:

1. Eine Variable ist kein Parameter. Dann wird sie gemäß ihrem Typ mit 0, 0.0 oder (Nullstring) initialisiert.
2. Die Variable ist ein Call-By-Value-Parameter. Es wird eine lokale Kopie dieser Variable erzeugt und mit dem Wert des Parameters initialisiert.
3. Die Variable ist ein Call-By-Reference-Parameter. Dann wird keine lokale Variable erzeugt, sondern im Environment eine Referenz auf die Originalvariable eingetragen.

4.4.2.2 ExecuteProfile

Pre
- Das Environment env ist aufgebaut und initialisiert.
- IP zeigt auf den ersten auszuführenden Befehl.
- Profilecommands ist eine Liste von Profilebefehlen.

Post Der Wert der Returnvariable ist bei Unterprogrammen (Sub) undefiniert.

```
ExecuteProfile(env:Environment,IP:Integer,
               ProfileCommands:Command*)
  repeat
    Command=ProfileCommands[IP]
    case Command of
      goto(NeuesLabel) : IP=NeuesLabel;
      condgoto(Bedingung, NeuesLabel) :
          if(Eval(Bedingung, env)<>0) then IP=NeuesLabel
                                      else IP=IP+1;
      default : begin
                  Execute(Command, env);
                  IP=IP+1;
                end;
   end;
  until Command ∈ {funreturn, subreturn}
     or IP>MaxCommandIndex;
  return env(ReturnVariable);
```

Zwei Kommandos werden in `ExecuteProfile` gesondert behandelt, nämlich `goto` und `condgoto`. Bei ersterem wird einfach der Befehlsindex IP auf einen neuen Wert gesetzt, bei zweiterem geschieht das nur, wenn eine Bedingung einen Wert ungleich 0 ergibt. Diese Sonderbehandlung ist aus zwei Gründen gerechtfertigt:

1. Wenn `ExecuteProfile` sie nicht gesondert behandelt, müsste IP eine globale Variable sein, da die beiden Befehle IP sonst nicht manipulieren könnten.

2. Wie im Defaultteil des `case`-Statements ersichtlich, wird IP nach jedem Befehl um eins (1) erhöht. Die genaue Definition von `goto` und `condgoto` wäre demnach abhängig, ob IP vor oder nach `Execute(Command,env);` erhöht wird.

4.4.2.3 eval(Ausdruck,env)

Pre
- Ausdruck entspricht der Syntax und den is-wf-... Regeln.
- Alle Variablen im Ausdruck sind auch im Environment (env) definiert.
- env ist das aktuelle Environment.

Inv Keine Variable des env wird verändert, außer ein Aufruf einer benutzerdefinierten Funktion (funcall) hat einen entsprechenden vom Benutzer programmierten Seiteneffekt.

Post Funktionsergebnis und Typ wird durch den Ausdruck im return-Stmt bestimmt.

```
eval(Ausdruck,env)
  case Ausdruck of
    var : return env(var.name);
    konst: return konst.value;
    (term) : return eval(term,env);
    t1 op t2 : case op of
                intplus  : return eval(t1,env)+eval(t2,env);
                intminus : return eval(t1,env)-eval(t2,env);
                intmal   : return eval(t1,env)*eval(t2,env);
                intdiv   : let x=eval(t2,env)
                           in if (x=0) then raise DivZero
                              else return eval(t1,env) div x;
                intmod   : let x=eval(t2,env)
                           in if (x=0) then raise DivZero
                              else return eval(t1,env) mod x;
                realplus : return eval(t1,env)+eval(t2,env);
                realminus: return eval(t1,env)-eval(t2,env);
                realmal  : return eval(t1,env)*eval(t2,env);
                realdiv  : let x=eval(t2,env)
                           in if (x=0) then raise DivZero
                              else return eval(t1,env)/x;
                strplus  : return concat(eval(t1,env),
                                         eval(t2,env));
                lessthan :
                  if eval(t1,env) < eval(t2,env)
                     then return 1
                     else return 0;
                lessequal:
```

```
                  if eval(t1,env) ≤ eval(t2,env)
                     then return 1
                     else return 0;
               equal    :
                  if eval(t1,env) = eval(t2,env)
                     then return 1
                     else return 0;
               greaterequal :
                  if eval(t1,env) ≥ eval(t2,env)
                     then return 1
                     else return 0;
               greater  :
                  if eval(t1,env) > eval(t2,env)
                     then return 1
                     else return 0;
               notequal :
                  if eval(t1,env) ≠ eval(t2,env)
                     then return 1
                     else return 0;
            end;
   op t2 : case op of
            intplus   : return  eval(t2,env);
            realplus  : return  eval(t2,env);
            intminus  : return -eval(t2,env);
            realminus : return -eval(t2,env);
         end;
   funcall(Info:TCallInfo,paramlist):
      return MakeFunCall(Info,paramlist);
end;
```

Hervorzuheben ist die Ähnlichkeit zwischen Funktions- und Unterprogrammaufruf, die sich nur dadurch unterscheiden, daß nach dem Funktionsaufruf in `eval` der Rückgabewert von `ExecuteProfile` weiterverwendet wird, während der (zufällige) Rückgabewert vom `call`-Befehl verworfen wird. Die Struktur *TCallInfo* beinhaltet die Nummer des ersten Befehls des Unterprogramms bzw. der Funktion und eine Beschreibung des Environments der Funktion bzw. des Unterprogramms.

4.4.2.4 MakeFunCall(Info:TCallInfo,paramlist)

Pre
- Info.start zeigt auf den ersten Befehl des Funktionskörpers
- Info.envDesc beschreibt das neue Environment (d.h. Name und Typ und Art (CallbyValue, CallbyReference) von Variablen)
- paramList beschreibt die Parameter als Referenz oder Ausdruck. Mit dieser Info wird in MakeNewEnvironment das neue Environment initialisiert.
- profileCommands ist eine (globale) Liste aller Befehle des Profiles

```
MakeFunCall(Info:TCallInfo,paramlist)
    return ExecuteProfile(
                MakeNewEnvironment(Info.envDescription,
                                   paramlist),
                Info.Start,
                ProfileCommands);
```

4.4.2.5 call(Info:TCallInfo,paramlist)

Pre
- Info.start zeigt auf 1. Befehl des Unterprogrammkörpers
- Info.envDesc beschreibt das neue Environment (d.h. Name und Typ und Art (CallbyValue, CallbyReference) von Variablen)
- paramList beschreibt die Parameter als Referenz oder Ausdruck. Mit dieser Info wird in MakeNewEnvironment das neue Environment initialisiert.
- profileCommands ist eine (globale) Liste aller Befehle des Profiles

```
call(Info:TCallInfo,paramlist)
  ExecuteProfile(MakeNewEnvironment(Info.envDescription,
                                    paramlist),
                 Info.Start,
                 ProfileCommands);
```

4.4.2.6 Funreturn Subreturn

```
funreturn(expr)
  env.ReturnVariable=Eval(expr,env);


subreturn
  skip
```

4.4.3 TCS-Befehle

Hier finden wir eine der vorhin erwähnten systemabhängigen Spezifikationen. Der Omega Controller besitzt 2 Betriebsmodi:

1. Angabe eines Setpoints (d.h. einer Zieltemperatur): Dabei wird nur ein Temperaturwert übertragen und der Controller beginnt sofort, diese Temperatur anzusteuern, wobei er nur von einem Tuneset und den physikalischen Eigenschaften des Geräteaufbaus und der Umgebung beeinflußt wird.

2. Verwenden eines Omega Profiles: Dabei läuft im Controller ein Programm ab. In diesem Programm kann eine Zeitvorgabe zum Erreichen der Temperatur und eine Toleranzgrenze angegeben werden. Der Controller geht dann erst zum nächsten Setpoint-Befehl, wenn sowohl die Zeitvorgabe als auch die Toleranzvorgabe erfüllt sind (Sofern eine Toleranz spezifizert wurde).

Diese beiden Modi finden sich auch im ramp-Befehl, welcher unterscheidet, ob die Angabe eines Setpoints allein reicht oder nicht.

4.4.3.1 ramp

Inv omegachannel ist geöffnet und die Verbindung steht

```
ramp(temp:TTemperatur,time:TTime,tol:REAL)
  if (time=0 and tol=0.0) then
  begin
    send(omegachannel,SetSetpoint(temp);
    while temp<>receive(omegachannel,GetTemperature) do
      skip;
  end else begin
             p=CreateProfile(temp,time,tol);
             send(omegachannel,WriteProfile(p));
             send(omegachannel,StartProfile);
             while (receive(omegachannel,
                    GetMode)=ModeExecute)) do
               skip;
             send(omegachannel,SetSetpoint(temp));
             send(omegachannel,SetMode(ModeExecute));
           end;
```

4.4.3.2 TSRead(tsname:STRING)

Aktuelles Tuneset wird aus einer Datei gelesen.

Pre tsname ist ein (Pfad +) Dateiname einer existierenden Tunesetdatei.

Inv omegachannel ist geöffnet und bereit.

Post Omega Controller verwendet neues Tuneset.

```
TSRead(tsname:STRING)
  if('.' ∉ tsname) tsname=tsname+'.SET';
  FILE f=open(tsname,read);
  Tuneset tuneset=read(f);
  send(omegachannel,WriteTuneset(tuneset));
  close(f);
```

4.4.3.3 TSWrite (tsname:STRING)

Aktuelles Tuneset wird in eine Datei geschrieben.

Pre tsname ist ein gültiger (Pfad +) Dateiname.

Inv • omegachannel ist geöffnet und bereit.
• Tuneset des Omega Controllers ändert sich nicht.

```
TSWrite (tsname:STRING)
  if('.' ∉ tsname) tsname=tsname+'.SET';
  Tuneset tuneset=receive(omegachannel,ReadTuneset);
  FILE f=open(tsname,write);
  write(f,tuneset);
  close(f);
```

4.4.3.4 TSPush

Inv • omegachannel ist geöffnet und bereit.
• Tuneset des Omega Controllers ändert sich nicht.

```
TSPush
  Tuneset tuneset = receive(omegachannel,ReadTuneset);
  push(TunesetStack,tuneset);
```

4.4.3.5 TSPop

Inv omegachannel ist geöffnet und bereit.

Post Wenn der Stack nicht leer war, verwendet der Omega Controller das neue Tuneset. Ansonsten wird das Tuneset des Omega Controllers nicht verändert.

```
TSPop
  if (not isempty(TunesetStack))
    send(omegachannel,WriteTuneset(pop(TunesetStack));
```

4.4.3.6 GetTemp

```
fun GetTemp
  return receive(omegachannel,GetTemperature);
```

4.4.4 Benachrichtigungsbefehle

4.4.4.1 Beep(SoundId:STRING,Soundintervall:INTEGER)

Pre Soundintervall ≥ 0

Inv Das Profile wird fortgesetzt, auch wenn keine Bestätigung des Users erfolgt.

```
Beep(SoundId:STRING,Soundintervall:INTEGER)
  repeat
    playsound(SoundId);
    sleep(SoundIntervall);
  until (SoundIntervall=0 or receive(UserInput)=StopSound);
```

4.4.4.2 Wait (Message:STRING,SoundId:STRING,Soundintervall:INTEGER)

Pre Soundintervall ≥ 0

```
Wait (Message:STRING,SoundId:STRING,Soundintervall:INTEGER)
  if(Message='') then
    Message='Mit einem Tastendruck geht es weiter.'
  send(UserOutput,Message);
  played=0;
  repeat
    if(SoundId<>'' and (SoundIntervall<>0 or played=0)) then
    begin
      playsound(SoundId);
      sleep(SoundIntervall);
      played=1;
    end;
  until (receive(UserInput)=Acknowledged);
```

4.4.5 Ein-/Ausgabebefehle

Im Folgenden ist *maxFiles* eine implementationsabhängige Konstante, die die Anzahl der gleichzeitig offenen Files angibt.

4.4.5.1 print(exprlist)

Pre exprList ist eine Liste gültiger Ausdrücke.

```
print(exprlist)
  let intostring([],s) = s
      intostring(e:el,s) = intostring(el,s+converttostring(e))
  in begin
       send(UserOutput,WriteString(intostring(exprlist)));
       while(receive(UserInput)<>Acknowledge) do
         skip;
     end;
```

4.4.5.2 Input(Prompt:STRING,Variable:Referenz auf Variable)

Post Variable besitzt neuen Wert.

```
Input(Prompt:STRING,Variable:Referenz auf Variable)
  if Prompt='' then Prompt=Standardprompt(gettype(Variable));
  send(UserOutput,WriteString(Prompt));
  repeat
    Answer=receive(UserInput);
    Letstry=Convert(Answer,gettype(Variable));
  until NoErrorWithConversion;
  Variable=LetsTry;
```

4.4.5.3 open(fname:STRING,fnr:INTEGER,mode:TMode)

Pre
- $0 < \text{fnr} \leq \text{maxFiles}$
- fname ist ein gültiger (Pfad +) Dateiname

Inv Verhalten von OSOpen wird durch Betriebssystem bestimmt.

Post Filetable[fnr] enthält Handle der geöffneten Datei.

```
open(fname:STRING,fnr:INTEGER,mode:{APPEND,INPUT,OUTPUT})
  if(Filetable[fnr]<>Unused) raise FileInUse;
  Filetable[fnr]=OSOpen(fname,mode);
```

4.4.5.4 close(fnr:INTEGER)

Pre $0 < \text{fnr} \leq \text{maxFiles}$

```
close(fnr:INTEGER)
  if(Filetable[fnr]<>Unused) OSClose(Filetable[fnr]);
  Filetable[fnr]=Unused
```

4.4.5.5 fileprint(fnr:INTEGER,exprlist)

Pre
- $0 < \text{fnr} \leq \text{maxFiles}$
- exprlist ist eine Liste gültiger Ausdrücke
- Filetable[fnr] identifiziert offene Datei.

```
fileprint(fnr:INTEGER,exprlist)
  let DoWork(e::[]) = OSWrite(Filetable[fnr],ConvertToString(e))
      DoWork(e::el) = OSWrite(Filetable[fnr],ConvertToString(e));
                      DoWork(el);
  in DoWork(exprlist);
  OSWrite(Filetable[fnr],EOL);
```

4.4.5.6 fileinput(fnr:INTEGER,var:Referenz auf Stringvariable)

Pre
- $0 < \text{fnr} \leq \text{maxFiles}$
- Filetable[fnr] identifiziert offene Datei.

```
fileinput(fnr:INTEGER,var:Referenz auf Stringvariable)
  var=OSRead(Filetable[fnr]);
```

4.4.5.7 eof(fnr:INTEGER)

Pre $0 < \text{fnr} \leq \text{maxFiles}$

```
fun eof(fnr:INTEGER)
  if(Filetable[fnr]=Unused) then return 1
                            else return OSEof(Filetable[fnr]);
```

4.4.5.8 delete(name:STRING)

Pre • name ist ein gültiger (Pfad +) Dateiname.
• Datei ist nicht geöffnet.

Post Datei ist gelöscht.

```
delete(name:STRING)
  OSDeleteFile(name);
```

4.4.5.9 getdir

```
fun getdir
  return OSGetWorkingdirectory;
```

4.4.5.10 chdir(name:STRING)

Pre name ist ein gültiger Verzeichnisname (+ Laufwerk)

Post aktuelles Verzeichnis ist entsprechend *name* geändert.

```
chdir(name:STRING)
  OSChangeDirectory(name);
```

4.4.6 Kontrollbefehle

Im Gegensatz zu den vorherigen Befehlen, welche eine eher kompakte Struktur haben, schließen sowohl IF-THEN-ELSE als auch WHILE-DO-WEND andere Befehle ein. Deshalb werden sie auch durch eine Kombination von bedingten und unbedingten Sprüngen realisiert.

```
IF <Expr> THEN S1;...;Sn ELSE Sn+1;...Sk ENDIF →
            condgoto(not (Expr),LabelSn1);
            S1
            .
            .
            Sn
            goto(LabelEndif);
LabelSn1    Sn+1
            .
            .
```

```
            Sk
LabelEndif


While <expr> DO S1;...;Sn WEND; →
LabelWhile condgoto(not(expr),LabelWend)
            S1
            .
            .
            Sn
            goto(LabelWhile);
LabelWend
```

Der Befehl `goto`, der sowohl in der Syntax als auch in der Semantik definiert wurde, wie auch `condgoto`, der für den User nicht zugänglich ist, wirken sich auf den Befehlsindex (IP) aus und werden deshalb in `ExecuteProfile` gesondert behandelt.

4.5 Anmerkung

Ein aufmerksames Lesen der Spezifikation der Profilesprache führt zur Erkenntnis, daß für eine Handvoll Befehle (TCS-Befehle und Benachrichtigungsbefehle), die benötigt werden, ein eigentlich unverhältnismäßig hoher Zusatzaufwand nötig ist, um eine ordentliche (special-purpose) Sprache zu definieren. Insbesondere die Ein-/Ausgabe- Befehle (ausgenommen der Benachrichtigungsbefehle) wurden nur deshalb in die Sprache aufgenommen, um einen Debugger zu vermeiden. Im konkreten Fall gab es aber keine vernüftige Alternative, da MS Windows keine einfache Scriptsprache beinhaltet, die es auch einem unerfahrenem User ermöglicht, ohne großen Aufwand ein passables Programm zu erstellen. Andernfalls hätte es gereicht, nur die benötigten Funktionen als Unterprogramme zu erstellen. Es empfielt sich daher, immer zu überprüfen, ob denn die Erstellung einer neuen Sprache auch sinnvoll und nötig ist.

5. IMPLEMENTIERUNG

5.1 Ereignisse

Hier die Definition von einem Ereignis (TEvent), wie es implementiert wurde:

```
TEvent = object (TObject)
           Temp:extended;
           Time:longint;
           Desc:array[0..DescSize] of char;
           constructor init(aTemp:extended; aTime:longint);
           constructor Load(var s:TStream);
           constructor copy(P:PEvent);
           procedure Store(var s:TStream);
           procedure SetDescription(newDesc:Pchar);
           procedure GetDescription(aDesc:PChar);
           function GetTime:longint;
           function GetTemp:extended;
         end;
```

Zum Vergleich dazu nocheinmal die Spezifikation eines Ereignisses von Seite 30:

```
TEREIGNIS =
class
  type TEreignis,TTemperatur,TDescription


  value
    create : TTemperatur × TDescription → TEreignis
    temp : TEreignis → TTemperatur,
    desc : TEreignis → TDescription
end
```

Zwei Unterschiede fallen sofort auf:

1. Die Spezifikation ist funktional, die tatsächliche Implementation aber objektorientiert.
2. Im Gegensatz zur Spezifikation gibt es eine weitere Eigenschaft in TEvent: Time.

Ad 1.) Es ist ohne Zweifel richtig, daß eine objektorientierte Implementation und eine objektorientierte Spezifikation geringere Reibungsverluste aufweisen würden. Es gibt keinen Philosophieunterschied und die Termini und Modelle der Spezifikation lassen sich direkt für die Implementation verwenden.

Andererseits sind diese Reibungsverluste nicht besonders groß und darüberhinaus sind beide Methoden gleichwertig bezüglich Genauigkeit und Vielfalt der Ausdrucksmöglichkeiten. Der eigentliche Grund für dieses Vorgehen liegt aber darin, daß zum Zeitpunkt des Beginns meiner Arbeit mir keine geeigneten OOP-Methoden zur Verfügung standen. [Dav90] beschreibt OOP nur am Rand und [Boo97] war mir damals unbekannt. Die funktionale Programmierung war mir aber aufgrund der Vorlesung Softwareparadigmen ziemlich geläufig.

Ad 2.) Wie bereits mehrmals erwähnt, ist es wichtig, daß die Spezifikation der Implementation nur in sehr gut begründeten Ausnahmen vorgreift und ansonsten nur das *Was* und nicht das *Wie* der zu implementierenden Funktionalität vorgibt. Die Begründung möchte ich an diesem Beispiel liefern:

Im vorligenden Fall muß die Ereignisliste sowohl in der Reihenfolge der Eingabe als auch sortiert nach Temperatur bearbeitet werden können. Dieses kann auf verschiedene Arten implementiert werden. Ich wählte ein Verfahren mit einem Timestamp bei der Erzeugung des Ereignisses. Sortiert man nach diesem Timestamp, ergibt sich die ursprüngliche LIFO- Ordnung. Da der Timestamp ansonsten nicht gebraucht wird, fehlt er in der Spezifikation, und man könnte auch einen anderen Lösungsweg verfolgen (z.B. über Pointer). Für diese Freiheit muß die Spezifikation aber abstrakt genug sein.

5.2 Windowsprogrammierung

Die Windowsprogrammierung ist relativ komplex. Aus diesem Grund gibt es verschiedene Bibliotheken, die die Programmierung vereinfachen, indem Routineaufgaben abgenommen werden. Für dieses Projekt wurde Borlands Object Windows Library (OWL) verwendet, welche mit Borland Pascal 7.0 ausgeliefert wird (siehe [192],[292]). Für ein besseres Verständnis der internen Zusammenhänge siehe [PN91].

5.2.1 Nachrichtenschleife

Jedes Windowsprogramm besteht im wesentlichen aus einer Hauptschleife (genannt Nachrichtenschleife), die (vereinfacht) folgendes Aussehen hat:

```
WHILE GetMessage(Msg,...) DO
BEGIN
  TranslateMessage(Msg);
  DispatchMessage(Msg);
END;
```

Da Windows in den Versionen bis 3.1 kein preemptives Multitasking unterstützt (zumindest nicht für Windowsprogramme), muß jedes Windowsprogramm dem Betriebssystem die Möglichkeit gewähren, ihm den Prozessor zu entziehen. Dies geschieht in der Funktion *GetMessage*. GetMessage ist solange

"wahr", bis das Programm beendet und eine *WM_QUIT* - Message von GetMessage erhalten wird. Wenn nun ein Programm in eine Schleife eintritt, muß es diese Nachrichtenschleife von Windows nachbilden. Dafür verwendet man anstelle von GetMessage die Funktion PeekMessage, die solange "wahr" ist, solange eine Nachricht in der Message-Queue vorhanden ist.

Aufgrund dieser zwei verschachtelten Schleifen ist schnell ersichtlich, daß ein Windowsprogramm innerhalb einer PeekMessage-Schleife nicht korrekt beendet werden kann (Die innere Schleife würde zwar beendet werden, aber die Hauptschleife wäre immer noch aktiv). Durch mehrfach verschachtelte Aufrufe, können auch Reentrance-probleme auftreten.

5.2.2 Allgemeines zu Windows Nachrichten

Windows 3.1 verfügt über eine systemweite Nachrichtenschlange. Aus dieser werden der Reihe nach die Nachrichten entnommen und an "ihr" Fenster übergeben.

5.2.3 Timernachrichten

Windows stellt eine sehr komfortable Menge von Timerfunktionen zur Verfügung. Dummerweise sind Timernachrichten von äußerst geringer Priorität. Timer sind (in der Regel) an Fenster gebunden. Timerereignisse werden nur an ihr Fenster übergeben, wenn keine andere Nachricht vorliegt. Weiters sind Timerintervalle im Bereich von Millisekunden möglich. Ihre Genauigkeit ist aber unvorhersagbar. Es können auch durchaus Timerereignisse verloren gehen. Jedesmal, wenn ein Timer zündet und keine andere Nachricht in der Queue vorhanden ist, wird von GetMessage bzw. PeekMessage eine WM_TIMER Nachricht zurückgeliefert.

5.3 Virtueller Omegacontroller

Um die Geräteabhängigkeit zu reduzieren, wurde das Objekt TOmega erstellt. Dieses versteckt für alle anderen Programmteile die Tatsache, daß eine Kommunikation mit dem Omega Controller stattfinden muß.

Dieses Objekt besitzt folgende Methoden:

init (aParent:PWindowsObject; aCommunication:PCommunication; aSecurity:PSecurity);

aParent gibt den Empfänger der Nachrichten an.

aCommunication das zu verwende Objekt Communication.

aSecurity die Sicherheitsparameter.

Initialisiert den Virtuellen Omega Controller.

done Keine Parameter. Stoppt den Omega Controller und restauriert dessen ursprüngliches Tuneset. Der Kommunikationskanal zum Omega Controller wird geschlossen und der Virtuelle Omega Controller de-Initialisiert.

WriteRamp (aSP:extended; aTime:longint; aTolerance:extended);

Wenn sowohl aTime als auch aTolerance gleich 0 sind, wird eine SET-POINT-Anweisung an den Omega Controller übermittelt. Ist auch nur einer dieser Parameter ungleich 0, so wird ein Mini-Profile generiert, welches an den Omega Controller gesendet und dort gestartet wird.

ReadSetpoint Liest den aktuellen Setpoint vom Omega Controller und liefert ihn kalibriert zurück.

ReadTemperatur Liest die aktuelle Temperatur und liefert sie kalibriert zurück.

ReadTunesetRaw (var aTuneset:TTunesetFrame) Liest das aktuelle Tuneset des Omega Controllers und speichert es in aTuneset.

WriteTunesetRaw (aTuneset:TTunesetFrame) Überträgt aTuneset als aktuelles Tuneset an den Omega Controller.

HasStopped Liefert True, wenn der Omega Controller im Stop-Zustand ist.

ExecutesProfile Liefert True, wenn der Omega Controller ein Profile ausführt.

InAutoMode Liefert True, wenn der Omega Controller im Automatik-Zustand ist, d.h. die Temperatur ohne Profile steuert.

TryToConnect Überprüft nach einem Verbindungsverlust, ob die Verbindung wiederhergestellt werden kann. Führt dabei einen Reset des Kommunikationskanals aus.

StopController (StopIt:Boolean) Wenn StopIt = TRUE, dann wird der Omega Controller in den Stop-Zustand versetzt. Ansonsten in den passenden Steuerungsmodus (Automatik oder Profile).

IsDisconnected liefert True, wenn die Verbindung zum Omega Controller unterbrochen ist.

StoreSPInfo (var aspi:TSetpoint) Speichert den aktuellen Setpoint (Setpoint, Zeitvorgabe und Toleranz) in aspi.

ReloadSPInfo (aspi:TSetPoint) Restauriert die Setpointinformation aus aspi.

SetCalibration (aCalibration:PCalibration) Übergibt an den Virtuellen Omega Controller das neue Kalibrierungsobjekt. Wenn NIL übergeben wird, wird keine Kalibrierung verwendet.

UsingCalibration liefert einen Zeiger auf das verwendete Kalibrierungsobjekt oder NIL.

5.4 Omega Anzeige

Die Anzeige besteht aus einem Bitmap, welches den Hintergrund darstellt und auch die Größe bestimmt. In dieser Anzeige wird ein auf dem System vorhandener Font für die Temperaturanzeige verwendet. Die Anzeige erfolgt immer in der Einheit Celsius.

Unter der Temperaturanzeige werden drei Leds dargestellt, die durch eigene Objekte realisiert wurden. Das Hauptfenster übernimmt die Aufgabe, diesem Objekt Informationsänderungen mitzuteilen.

5.5 Kommunikation

5.5.1 Frames

Da dem Omega Controller mehrere Kommandos auf einmal gegeben werden können, die er der Reihe nach ausführt, wurden zwei Objekte entwickelt, die das Zusammenstellen des Command-Frames und die Interpretation des Antwortframes übernehmen.

Jede Methode dieser Objekte entspricht einer Kommandovariante. Der Kommandoframe baut den Frame in der Reihenfolge der Methodenaufrufe auf. Der Antwortframe muß in der gleichen Reihenfolge gelesen werden. Notwendige Formatumwandlungen werden von den einzelnen Methoden übernommen.

5.5.2 Communication

Das Communication-Objekt übernimmt die eigentliche Übertragung. Seiner Methode *Communicate* werden Zeiger auf einen Outputframe (=Kommandoframe) und einen Inputframe (=Antwortframe) übergeben. Communicate bildet den passenden Header und die Prüfsummen und sendet den Outputframe. Danach wartet es auf die Antwort, wobei ein Timeout-Mechanismus verwendet wird. Da die reguläre Antwortzeit recht kurz ist, wird darauf verzichtet, einem anderen Windowsprogramm die CPU abzutreten.

Nach Erhalt einer Antwort wird die Prüfsumme berechnet und überprüft. Im Falle eines Übertragungsfehlers oder eines Timeouts wird eine Fehlermeldung zurückgegeben.

5.6 Profile

Ein Profile ist eine Liste von Anweisungen. Jede Anweisung besteht aus einem Objekt, welches alle notwendigen Informationen besitzt, um sich selbst auszuführen. Die einzigen Objekte, die sich nicht selbst ausführen können, sind Sprungbefehle.

Die Anweisungen sind aufsteigend durchnummeriert. Bei einem Profilestart wird die "Startadresse" der Execute-Unterroutine mitübergeben. Nach jeder Anweisung wird der Instructionpointer um eins (1) erhöht, und zeigt damit auf die nächste Anweisung. Sprunganweisungen werden speziell behandelt und liefern die Adresse der nächsten Anweisung.

5.6.1 Variablen

Das Hauptprogramm und jedes Unterprogramm besitzen einen getrennten Variablenraum. Das heißt, es gibt keine globalen Variablen. Jede Variable ist ein eigenes Objekt. Bei der Initialisierung dieser Umgebung wird zwischen Call-by-Reference und Call-by-Value-Parameter unterschieden. Call-by-Value-Parameter werden als Objekt neu angelegt und initialisiert. Call-by-Reference-Parameter hingegen werden selbst in den neuen Variablenbereich übergeben (bzw. ein Zeiger auf sie).

5.6.2 Unterprogramme

Bei einem Unterprogrammaufruf wird zuerst der Variablenbereich für das Unterprogramm angelegt. Danach wird der Unterprogrammaufruf durch einen rekursiven Aufruf des Prozedur Execute realisiert.

5.6.3 Funktionen

Funktionen werden auf die gleiche Art wie Unterprogramme behandelt, mit dem Unterschied, daß sie nur bei der Auswertung von Ausdrücken auftreten können.

5.7 Direkteingabe

Die Direkteingabe der Temperaturwerte wird durch einen nicht-modalen Dialog realisiert. Die Kommunikation mit dem Omega Controller erfolgt durch das Senden von Nachrichten an das Hauptfenster.

5.8 Kalibrierungstabelle

Die Kalibrierungstabelle wird als sortierte Collection realisiert. Zum Einlesen der Werte wird der Scanner des Profile-Compilers verwendet, wobei nur Zahlen und die Operatoren + und - weiter verarbeitet werden und jedes andere Token einen Fehler bedeutet, der dazu führt, daß keine Kalibrierung verwendet wird.

Die Kalibrierungstabelle besteht aus Records mit den Werten T und TCal, wobei TCal der kalibrierte Wert und T der vom Omega Controller gemessene Wert ist.

6. ZUSAMMENFASSUNG

6.1 Rückblick

Ausgangspunkt dieser Arbeit war der Wunsch nach einem besseren Userinterface für den Omega Controller, einem Gerät das zur Temperatursteuerung benötigt wird, um Einschlüsse in Gesteinsdünnschliffen zu untersuchen. Von diesem Wunsch ausgehend wurde eine Analyse des Istzustandes durchgeführt, in der festgestellt wurde, welche Aufgaben an und für sich durchfegführt werden und welche Aufgaben davon durch eine Softwarelösung ersetzt, verbessert oder bedienerfreundlicher gemacht werden können. Diese Aufgaben wurden dann in den Requirements konkretisiert, indem festgehalten wurde, welche Aufgaben wie zu erledigen sind. Dieses *Wie* wurde danach in der Formalen Spezifikation festgehalten. Diese Spezifikation war dann die Basis für die Implementierung. Ziel und Umfang dieser Arbeit war es also, den Entwicklungsprozeß eines Softwareprojektes von der Analyse des Ist-Zustandes über das Suchen der Requirements, ihrer Spezifikation bis zu deren Implementation zu verfolgen und zu kommentieren.

6.2 Ausblick

Folgende Arbeiten hätten noch durchgeführt werden können (und warum sie nicht gemacht wurden):

1. Einsatz einer DLL, um Controller-unabhängig zu sein.

2. Anstelle den Temperaturverlauf durch Soll-Temperatur, Zeit, Toleranz und Tuneset zu spezifizieren, eine abstraktere Methode finden.

   Diese beiden Punkte stehen in einem sehr engen Zusammenhang. Tatsache ist aber, daß der Omega Controller bereits am Institut vorhanden war, und daher die Benutzer bereits mit der Trennung in Zeit, Temperatur, Tuneset und Toleranz sehr vertraut waren. Da desweiteren ein Austausch des Controllers nicht beabsichtigt ist, jedoch eine schnelle Verfügbarkeit des Programms erwünscht war, gab es keinen Bedarf, diese Geräteunabhängigkeit zu verwirklichen.

3. Einsatz eines Framegrabbers:

Ein Framegrabber stand leider nicht zur Disposition, da der Videoprinter bereits geordert war. Abgesehen davon, daß mit Einsatz eines Framegrabbers ein Monitor eingespart und die Bearbeitung der Bilder und deren Einbindung in einen Ergebnisbericht möglich geworden wäre, gäbe es eine Reihe von Möglichkeiten das System zu erweitern:

(a) Zeitreihenbild

Man könnte die gesamte Untersuchung bildlich festhalten und, anders als bei einem Videorekorder, direkt auf einzelne Frames (evt. mit Zeit/Bildindex oder ähnlichem) zugreifen. Es wäre ein leichtes, Zusatzdaten wie z.B. die aktuelle Temperatur etc. einzublenden.

(b) Computer übernimmt die Beobachtung der Probe

Es wäre zu überprüfen, ob Methoden der Bildanalyse den Computer in die Lage versetzen können, die Untersuchung der Probe (weitestgehend) selbständig zu übernehmen. Natürlich müßte man sich auch fragen, ob diese Lösung im Rahmen eines Ausbildungsbetriebes überhaupt sinnvoll ist. Außer der Frage nach dem Sinn gibt es eine praktische Schwierigkeit: Durch die doch recht starken Temperaturveränderungen entsteht Bewegung in der Probe. Dadurch muß entweder das Mikroskop neu fokussiert werden oder die Probe selbst muß verschoben werden (manchmal beides). Beide Vorgänge sind mit der gegenwärtigen Konfiguration nicht durch den Computer erledigbar. Deshalb müsste immer ein Mensch anwesend sein, der die Probe beobachtet und der kann dann auch gleich die Untersuchung durchführen.

4. Einsatz eines Formulargenerators

Damit könnte die Erstellung eines Berichts unterstützt werden, insbesondere, wenn bereits die Bilder des Framegrabbers zur Verfügung stehen. In der gegenwärtigen Situation muß der Experimentator allerdings eine Skizze der Probe erstellen (können), deren Einbindung in den computergestützen Bericht zumindest sehr aufwendig und mit zusätzlichen Kosten verbunden wäre (Scanner).

5. Unterstützung durch Datenbanken

(a) Eine Datenbank, die zu Ereignissen eine Erklärung findet.

Hier stellt sich wieder die Frage, ob ein solches Feature im Rahmen eines Systems zur Ausbildung sinnvoll ist. Im gegenwärtigen Fall wurde dies verneint.

(b) Eine Datenbank über bereits erfolgte Untersuchungen

Sinnvollerweise wäre hier der Einsatz des Formulargenerators eine Voraussetzung, um nicht die gleichen Daten mehrmals erfassen zu müssen.

6.3 Nachbetrachtungen

Ich habe, aufgrund einiger zum Teil nicht vermeidbarer Umstände, mit dieser Arbeit länger gebraucht als ursprünglich geplant war. Sowohl Mag. Loizenbauer, meine direkter Ansprechpartner auf der KFUG als auch ich selbst wurden (leider nicht gleichzeitig) zum Zivildienst eingezogen. Dieser Umstand führt aber nun dazu, daß ich in der Lage bin, zu überprüfen, wie sich das TCS in der Praxis bewährt hat. Das für mich erfreuliche Ergebnis ist, daß das TCS seine Aufgaben sehr but erfüllt und noch immer im Einsatz ist. Im Augenblick wird sogar über eine Folgeversion nachgedacht, da einige Punkte aus dem vorigen Abschnitt realisiert werden sollen (Einsatz einer Videograbber-karte und die automatische Analyse der Einschlüsse). Trotzdem gibt es zwei Sackgassen des Designs, die sich in der Praxis leider nicht so bewährt haben, wie es ursprünglich von mir gedacht war.

1. Von der Profilesprache werden im wesentlichen nur der Befehl `ramp` und die Benachrichtigungsbefehle verwendet. All die anderen Features wie Unterprogramme, Variablen, Berechnungen, ganz zu schweigen von den übrigen I/O-Befehlen, werden nicht verwendet. Der Grund dafür liegt in der Tatsache, daß ich offensichtlich einen komplexeren Temperaturverlauf für die Untersuchung für notwendig hielt. Ich habe es verabsäumt, zu überprüfen, ob *meine* Vorstellung eines Temperaturverlaufes mit der Vorstellung des Kunden übereinstimmt.

2. Die graphische Darstellung des Temperaturverlaufes ist zwar recht schön, in der Praxis aber nicht sehr brauchbar. Auch dies liegt in der Tatsache, daß der Temperaturverlauf in der Praxis zu einfach ist. Auch das Abspeichern und spätere Laden und darstellen bei einer zusätzlichen Untersuchung ist in der Praxis nicht notwendig, da die Untersuchung die Probe nicht nachhaltig verändert, es daher unwesentlich ist, ob bereits eine Untersuchung auf dieser Probe durchgeführt wurde oder nicht.

Glücklicherweise beeinflussen diese beiden Designirrtümer die Brauchbarkeit des Programms nicht im geringsten.

ANHANG

A. PID-STEUERUNG

Der **P-Wert** (PROPORTIONAL) wird in Einheiten der Prozeßvariable angegeben und ist der Kehrwert der Controller-Steigung, welche die Antwort des Controllers auf eine Veränderung am Ist-Temperatureingang steuert. Ein zu großer Anstieg (niedrieger P-Wert) veranlaßt das System zu oszillieren und instabil zu werden, während durch einen zu geringen Anstieg (hoher P-Wert) der Controller träge wird und zu lange braucht, um den Sollwert zu erreichen. Gültige Werte: 5-4000 Grad C

Der **I-Wert** (meist RESET genannt) paßt den Ausgang automatisch an die Abweichung zwischen Ist- und Soll-Temperatur an. Er wird in Wiederholungen pro Minute angegeben. Bei jeder Wiederholung wird die Ist-Temperatur in Richtung zum Sollwert verändert. Ist der Wert zu hoch, wird der Sollwert immer wieder verfehlt. Ist der Wert hingegen zu niedrig, dann wird das Erreichen des Sollwerts verzögert. Gültige Werte: 0.00 bis 20.00 Wiederholungen/Minute (0 schaltet die Steuerung aus)

Der **D-Wert** (meist RATE genannt) verursacht eine zeitlich beschränkte Änderung der Controller-Steigung als Antwort auf eine Änderung des Gleichgewichtszustands, welche nach einer programmierten Zeitperiode wieder zur Originalsteigung abklingt. Ist der D-Wert zu groß, kann der Controller den Soll-Wert verfehlen, ist er jedoch zu klein, kann es das Erreichen des Sollwerts verzögern. Gültige Werte: 0.00 bis 20.00 Minuten (0 schaltet die Steuerung aus)

B. EIN BEISPIEL FÜR EIN EXPERIMENT

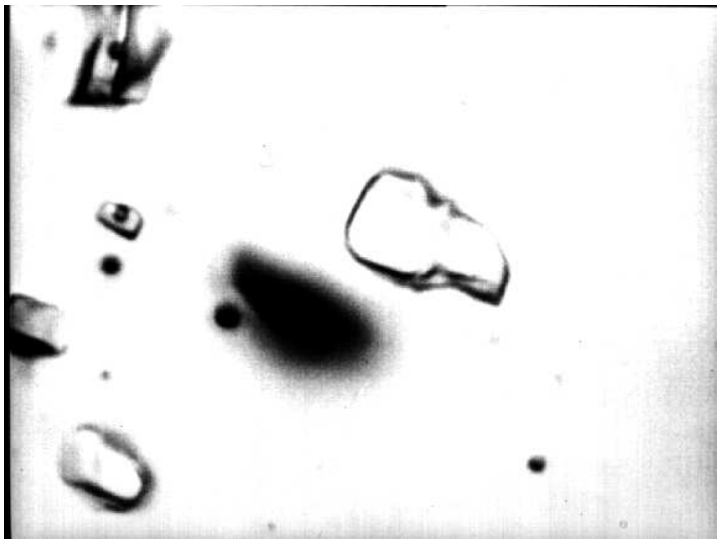

Abb 5 Quarzkristall bei 19 Grad Celsius

In Abbildung 5 sieht man eine Vergrößerung eines Hohlraumes in einem Quarzmineral. Die Aufnahme erfolgte bei 19 Grad Celsius. Dieser Hohlraum kann eine Flüssigkeit (und / oder eine Gasphase) enthalten und wird deshalb als Flüssigkeitseinschluß bezeichnet. Interessant ist der Flüssigkeitseinschluß, der auf dem Bild hell erscheint (Zu sehen im oberen rechten Quadranten des Bildes nahe der Bildmitte). Ziel der Untersuchung ist es, Informationen über diesen Flüssigkeitseinschluß zu erhalten (z.B. befinden sich darin gelöste Salze, und wenn ja, welche?).

Abb 6 Quarzkristall bei -70 Grad Celsius

Abbildung 6 erfolgte bei -70 Grad Celsius. Im Flüssigkeitseinschluß sind nun Eiskristalle sichtbar (die dunklen und hellen Flächen). Aus dem *Liquid* wurde nun eine *feste Phase*. Zu diesem Zeitpunkt beginnt die mikrothermometrische Untersuchung, welche nur während eines Heizvorganges erfolgt.

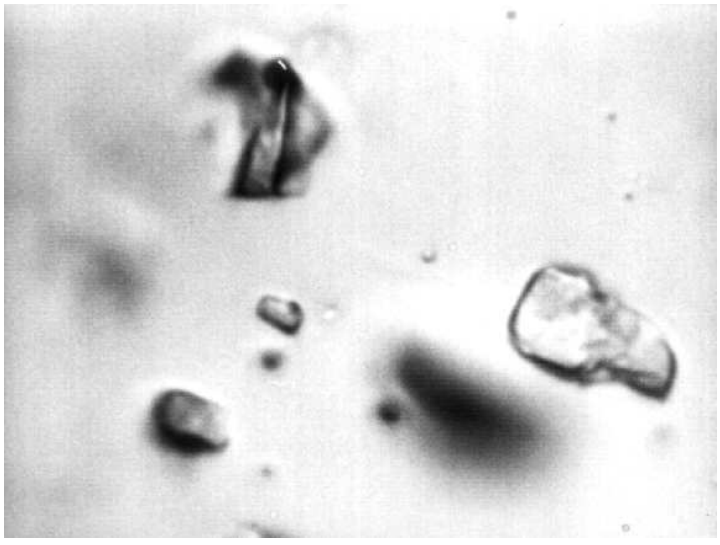

Abb 7 Quarzkristall bei -24 Grad Celsius

Abbildung 7 erfolgte bei -24.0 Grad Celsius. Während des langsamen Aufheizens, welches durch das TCS kontrolliert wird, beginnt die Salzlösung zu schmelzen. Der Beginn des Schmelzvorgangs wird *Eutektikum* genannt, und tritt in einem Mehrkomponentensystem bei einer systemspezifischen Temperatur ein (Dieser Zeitpunkt ist eines dieser *Ereignisse*, die protokolliert werden).

Aus der Kenntnis dieser Temperatur folgt die Zusammensetzung des Flüssigkeitseinschlusses. Jedoch ist noch immer unbekannt, wie hoch die Salzkonzentration ist und wieviel davon vorhanden ist.

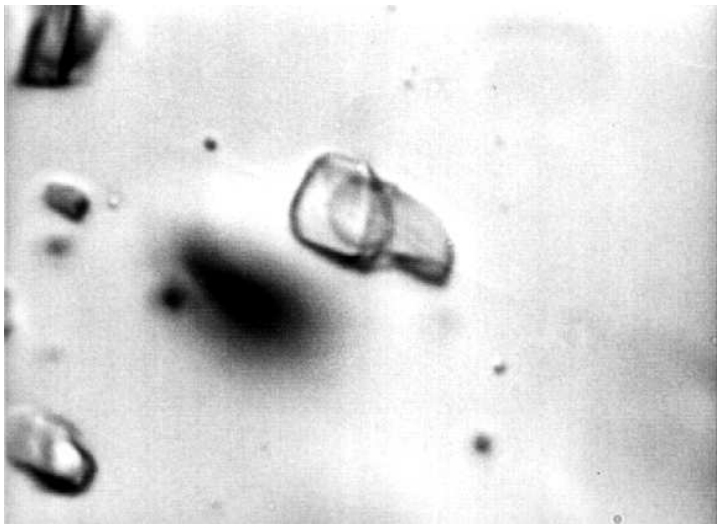

Abb 8 Quarzkristall bei -12 Grad Celsius

Abbildung 8 erfolgte bei -12 Grad Celsius. Der Schmelzprozeß läuft weiter fort, und immer mehr Eis geht in Lösung. Das kreisartige Gebilde im Einschluß ist das Eis.

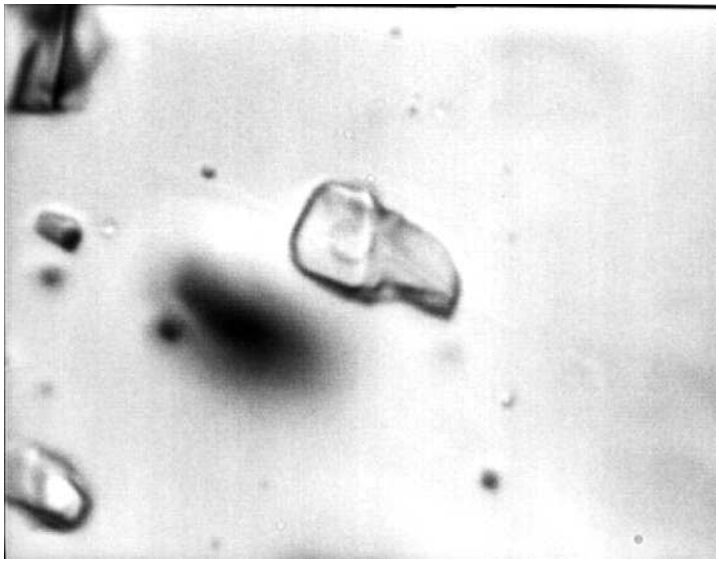

Abb 9 Quarzkristall bei -9.8 Grad Celsius

Wie aus Abbildung 9 (bei -9.8 Grad Celsius) ersichtlich, wird das Eisgebiet immer kleiner. Sobald das Eis vollständig verschwunden ist, wird wieder die

Temperatur protokolliert (ein weiteres *Ereignis*, welches *letztes Schmelzen* genannt wird). Daraus ergibt sich die Salzkonzentration der Lösung und wie hoch deren Konzetration ist.

Mit dem Wissen, welche und wieviel Salzkonzentration eingeschlossen wurde, ist die chemische Untersuchung abgeschlossen. In der Regel befinden sich aber auch Gase in den Einschlüssen. Dann beginnt der physikalische Teil der Untersuchung. Man erhitzt und beobachtet die Probe weiter, bis die Gasphase verschwindet. Dieser Punkt wird *Homogenisierung* genannt. Dadurch erfährt man, welche Temperatur und welcher Mindestdruck bei der Bildung des Einschlusses herrschten.

Ergebnis der Untersuchung

Eutektikum -24 Grad Celsius, das entspricht einer Natriumchlorid- Kaliumchlorid-Wasserlösung (NaCl-KCl-H_2O).

letztes Schmelzen bei -9.6 Grad Celsius, das entspricht einer 12 gewichtsprozentigen NaCl-KCl-K2O-Lösung, das heißt, daß 12 Prozent des Gesamtgewichts auf NaCl-KCl entfallen.

C. DER ARBEITSPLATZ

Abb 10 Arbeitsplatzübersicht

Abbildung 10 zeigt den Arbeitsplatz. In der oberen Reihe sieht man rechts den Monitor des Computers und links davon den Videoprinter. In der unteren Reihe ist außen rechts der Computer und links davon der Monitor für die Mikroskopkamera. In der Einheit daneben findet man rechts unten den Omega Controller und links unten den Chaimexa 2 Punktregler. Darüber steht der Videorekorder. Ganz links steht das Mikroskop.

Abb 11 Mikroskop mit Heiz-/Kühltisch

Bild 11 zeigt das Durchlichtmikroskop der Marke Olympus Typ PH2. Der kleine Messingring mit dem Schlauchanschluß ist der Heiz-/Kühltisch. Am oberen Ende des Mikroskops ist der Kameraaufsatz zu erkennen.

Abb 12 Chaimexa 2 Punktregler

An das Mikroskop ist der Chaimexa 2 Punktregler aus Bild 12 angeschlossen.

Abb 13 Omega Controller

Bild 13 zeigt die Frontansicht des Omega Controllers. Der regelt unter Verwendung des Chaimexa 2 Punkt-Reglers (welcher zu einer reinen Steuerung "degradiert" wurde) die Temperatur des Heiz-/Kühltisches.

Abb 14 Video Printer

Das Bild, das die Kamera liefert, kann mit dem Video Printer aus Bild 14 ausgedruckt werden. Wäre dieser Printer nicht bereits geordert gewesen, hätte man Methoden der Bildverarbeitung in das TCS einbinden können.

Abb 15 Bei der Arbeit

Während die vorherigen Bilder nur der Illustration dienen, wird in dieser Aufnahme (Bild 15) deutlich, warum ein akustisches Feed Back im Programm eingebaut wurde. Während einer genauen Untersuchung muß der Untersucher die Probe selbst beobachten. Die Verwendung des Monitorbildes der Kamera reicht nicht immer aus, da deren Blickfeld eingeengt ist.

LITERATURVERZEICHNIS

[192] Borland GesmbH. *Borland Pascal 7.0 Object Windows Programmierhandbuch*, 1992.

[292] Borland GesmbH. *Borland Pascal 7.0 Windows API Band 1 und 2*, 1992.

[Bil93a] Pat Billingsley. 1990 ec directive may become driving force. *SIGCHI Bulletin*, January, 1993.

[Bil93b] Pat Billingsley. A closer look at iso 9241. *SIGCHI Bulletin*, July, 1993.

[Bil93c] Pat Billingsley. The emergence of iso 9241 and some of its parts. *SIGCHI Bulletin*, April, 1993.

[BJ78] Dines Bjørner and C.B. Jones, editors. *The Vienna Development Method: The Meta Language*. Springer Verlag, 1978.

[Boo97] Grady Booch. *Objektorientierte Analyse und Design*. Addison-Wesley, 1997.

[Cal94] Gene Callahan. Excessive realism in gui design: Helpful or harmful? *Software Development*, 1994.

[Dav90] Alan M. Davis. *Software Requirements Analysis & Specification*. Prentice Hall, 1990.

[Geo95] Chris George. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1995.

[HR81] Christos H. Papadimitriou Harry R.Lewis. *Elements of the Theory of Computation*. Prentice Hall, 1981.

[JS90] Cliff B. Jones and Mary Shaw, editors. *Case Studies in Systematic Software Development Using VDM*. prentics-Hall, 1990.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications*. Wiley Professional Computing. John Wiley & Sons Ltd., 1992.

[OMEa] OMEGA Technologies. *CN3000 Serial Communications Protocol*.

[OMEb] OMEGA Technologies. *Dual Input Programmable Process Controllers*.

[PN91] Paul Yao Peter Norton. *Windows 3 Perter Norton's Programmiertechniken fuer Profis*. Markt und Technik Verlag, 1991.

[PvKT92] Nico Plat, Jan van Katwijk, and Hans Toetenel. Application and benefits of formal methods in software developement. *Software Engineering Journal*, 7(5):335–346, September 1992.

[RU93] Robert W. Root and Kathy M. Uyeda. A headsup on gui styleguides: Report on the chi'92 styleguide sig. *SIGCHI Bulletin*, July, 1993.

[Ten81] R.D. Tennent. *Principles of Programming Language*. Series in Computer Science. Prentice Hall International, 1981.

[vFH91] Foley vanDam Feiner Hughes. *Computer Graphics Principles and Practice, Second Edition*. Addison-Wesley, 1991.

[Wat91] David A. Watt. *Programming Languages Syntax and Semantics*. Series in Computer Science. Prentice Hall, 1991.

[Win90] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, 23(9):8–24, September 1990.

[Wix93] Dennis Wixon. Review of: Principles and guidelines in software user interface design. *SIGCHI Bulletin*, 1993.