Hochschule für Musik Franz Liszt Weimar
Studio für elektroakustische Musik (SeaM)
Fachrichtung Komposition

Diplomarbeit
zur Erlangung des akademischen Grades Diplom-Musiker (Komposition)

Arno

Über den Einsatz eines Suchalgorithmus für die musikalische Komposition

vorgelegt von Torsten Anders

Weimar, 3. Januar 2000

Mentoren:
Prof. Robin Minard
(elektroakustische und computergestützte Komposition, Weimar)
Prof. Dr. Clemens Beckstein
(Künstliche Intelligenz, Jena)

Das Programm Arno ist ein Werkzeug zur computergestützten musikalischen Komposition. Mit diesem Werkzeug erzeugt ein Komponist eine Partitur, indem er das gesuchte Ergebnis beschreibt. Der Komponist definiert Constraints — Eigenschaften, die das Ergebnis erfüllen soll. Das Programm sucht ein Ergebnis, das allen geforderten Constraints genügt.

Arno wurde im Rahmen dieser Arbeit von Autor entwickelt. Der folgende Text führt nach einer Einleitung (Kap. 1) in den theoretischen Hintergrund der verwendeten Programmiertechnik, das Programmieren mit Constraints, ein (Kap. 2). Arno erweitert Common Music, eine Umgebung zur computergestützten Komposition: das folgenden Kapitel stellt Common Music vor, zusammen mit der verwendeten Programmiersprache Common Lisp und einer Lisp-Erweiterung zur Constraints-Programmierung (Kap. 3).

Der zweite Teil der Arbeit handelt vom Programm Arno selbst: der Text demonstriert die Anwendung und die Möglichkeiten von Arno (Kap. 4) und erläutert implementatorische Details (Kap. 5). Abschließend werden verschiedene mögliche Verbesserungen zur Effizienzsteigerung und Erweiterungen der Leistungsmerkmale diskutiert (Kap. 6).

Inhaltsverzeichnis

1. Einleitung **5**

1.1. Motivation der computergestützten Komposition 5

1.2. Kreatives oder wissenschaftliches Interesse 5

1.3. Musik aus Zahlen 6

1.4. Paradigmen der computergestützten Komposition 6

2. Constraints **8**

2.1. Idee 8

2.2. Nicht-deterministische Programmierung 9

2.3. Algorithmen 10

2.4. Musikalische Projekte, die Constraints einsetzen 18

3. Verwendete Umgebung zur Verwirklichung von Arno **22**

3.1. Common Lisp 22

3.2. Common Music (CM) 23

3.3. Screamer 27

3.4. Suche nach geeigneten Mitteln 29

3.5. Zusammenspiel der verschiedenen Umgebungen 30

4. Die Anwendung von Arno **32**

4.1. Motivation 32

4.2. Ein erstes Beispiel 33

4.3. Das Leistungsprofil von Arno 35

4.4. Ein komplexeres Beispiel 37

5. Die technische Realisierung von Arno **49**

5.1. Einleitung 49

5.2. Das Makro `defcontain` 49

5.3. Erweiterungen der Common Music API 54

6. Offene Fragen **56**

6.1. Herausforderung: Komponieren durch Beschreiben 56

6.2. Erweiterungen zur Effizienzsteigerung 56

6.3. Weitere Erweiterungsmöglichkeiten 59

7. Fazit **61**

A. Lisptechniken **64**

A.1. Makros in Lisp 64

A.2. CLOS, objektorientierte Programmierung mit Common Lisp 65

A.3. Nondeterminismus in Lisp 66

B. Klassischer Kontrapunkt: ein ausführlicheres Beispiel **70**

C. Lispcode von Arno **79**

C.1. Die Datei *\arno-kernel.lisp: 79

C.2. Die Datei *\object-utils.lisp 84

C.3. Die Datei *\utils.lisp 92

C.4. Die Datei *\envelope.lisp 99

C.5. Die Datei *\envelope-object.lisp 104

C.6. Die Datei *\in-cm-package.lisp 111

C.7. Die Datei *\def-n-init-arno.lisp 111

1. Einleitung

1.1. Motivation der computergestützten Komposition

Musiker verwenden einen Computer für verschiedene Anwendungen: es können digitalisierte Klangaufnahmen vielfältig bearbeitet werden, Klänge lassen sich verschiedentlich synthetisieren und auf viele Weisen können Steuerinformationen (Partituren, die Klangbearbeitung oder -synthese steuern oder traditionelle Instrumentalpartituren) erzeugt werden. Die Erzeugung solcher Steuerinformationen, die computergestützte Komposition, ist Gegenstand dieser Arbeit.

Für einen Komponisten ist die computergestützte Komposition interessant, da sich musikalische Entwicklungen bzw. Prozesse unter Umständen leichter algorithmisch beschreiben, als ausnotieren lassen. Ein Algorithmus bietet einen höheren Grad der Abstraktion. Eine durch ein Programm erzeugte Entwicklung ist auch leichter zu verändern: geändert wird nur das Programm, d.h. die abstrakte Beschreibung und nicht eine Vielzahl einzelner Elemente (z.B. Noten).

Durch die Arbeit mit dem Rechner ist ein anderer Grad an Komplexität der Musik möglich. Es sind Datenmengen und -formen zu bewältigen, die „von Hand“ nur sehr mühsam geschrieben werden können. So könnten z.B. rhythmische Beziehungen oder eine mikrotonale Harmonik leichter algorithmisch zu definieren sein. Die kompositorische Verwendung der riesigen Datenmenge einer spektralen Analyse ist nur mit dem Computer vorstellbar.

Ist nicht eine traditionelle Instrumentalpartitur das Ziel, sondern die Steuerung von Soundsynthese- bzw. -bearbeitungsprozessen, wird häufig eine wesentlich größere Datenmenge benötigt. Eine traditionelle Partitur wird durch den ausführenden Musiker bei der Interpretation um sehr viele Informationen ergänzt. Eine Soundsynthese „spielt“ eine übergebene Partitur wie sie ist. Unter Umständen kann das deshalb notwende Vielfache an Information leichter algorithmisch erzeugt werden.

Durch computergestützte Komposition können schnell viele Varianten erzeugt werden, die hörend miteinander verglichen werden können.

Schließlich ist die Beschäftigung mit dem Kompositionsvorgang selbst ein in der europäischen Kunstmusik immer wieder behandeltes Thema. Der Musiktheorie (als Wissenschaft, nicht als pädagogische Disziplin) sind mit dem Einsatz eines Computers neue Wege der Analyse eröffnet. Durch algorithmische Komposition können deren Ergebnisse auch verifiziert werden.

1.2. Kreatives oder wissenschaftliches Interesse

Es lassen sich bei der Verwendung von Rechnern zur musikalischen Komposition zwei Grundtendenzen erkennen: bei der *computergestützten* Komposition wird der Computer als kreatives Werkzeug eingesetzt, um neue, bislang nicht oder nur äußerst aufwendig erreichbare musikalische

Ergebnisse zu erzielen. Bei der *automatisierten* Komposition dagegen wird im wissenschaftlichen Kontext an der Formulierung neuer leistungsfähigerer Algorithmen gearbeitet, zur Verifikation werden dabei gern allgemein bekannte traditionelle Kompositionstechniken eingesetzt. Die Motivation der jeweiligen Arbeit und damit auch die Herangehensweisen ist verschieden. Der Komponist sucht ein Kompositionsinstrument: ästhetische Entscheidungen fällt der Komponist, untergeordnete Details vollzieht ein Programm. Der Wissenschaftler will mit neuen Mitteln traditionelle Satztechniken analysieren und synthetisieren. Dieser ist vorranging am musikalischen Output, jener am Algorithmus interessiert — wobei natürlich Überschneidungen möglich sind.

1.3. Musik aus Zahlen

Mit dem Computer zu komponieren heißt, Zahlen zu generieren. In der seriellen Komposition war es üblich, ein musikalisches Ereignis (wie etwa eine Note) mit vielen *Parametern* zu beschreiben: eine Note kann z.B. eine Einsatzzeit, eine Dauer, eine Lautstärke und eine Tonhöhe besitzen. Die Grenzen dieses Ansatzes sind sehr früh erkannt wurden[1] und die Kompositionsgeschichte verfolgte bald wieder völlig andere Wege. In der computergestützten Komposition ist die Idee, eine Partitur mit numerischen Werten zu beschreiben, beibehalten bzw. wieder aufgegriffen worden, um die Partitur im Computer darstellen, bearbeiten und speichern zu können.

Mit numerischen Parametern lassen sich sehr genau Steuerinformationen zur Soundsynthese darstellen und Klangbearbeitungen können mit Hilfe von Parametern beschrieben werden. Eine Kompositionsumgebung mag eine abstrakte Partiturrepräsentation der musikalischen Struktur bieten, die darin enthaltenen Informationen sind numerische Parameterlisten. Letzlich kann ein Computer nur mit Zahlen umgehen, deshalb muß auch eine Partitur mit Zahlen ausgedrückt werden.

Numerische Parameter sind jedoch absolute Angaben. Es gibt beispielweise keine relativen Maße wie *piano* und *forte* (die zudem außer leise bzw. laut auch *sachte, langsam* bzw. *stark, groß* bedeuten). Mit numerischen Parametern dargestellte musikalische Gestalten und Qualitäten sind zum andern quantisiert, unter Umständen ist das Raster der Quantisierung nicht fein genug. Insbesondere aber ist das musikalische Denken dadurch beeinflußt: in der computergestützten Komposition werden nicht (direkt) musikalische Gestalten, ja nicht einmal Töne beschrieben. Auch wenn ein Programm eine höhere Form der Abstraktion bietet, tatsächlich geht der Komponist mit einzelnen Parametern um.

1.4. Paradigmen der computergestützten Komposition

In den letzten Jahrzehnten wurden verschiedene Strategien für die computergestützte Komposition entwickelt (Einsatz stochastischer Prozesse oder verschiedener Automaten, Fraktale und Chaos-Generatoren, generative Grammatiken usw., eine Überblick bietet Roads [1996, Kapitel 18f]). Die Techniken sind vielfältig: die Entscheidung für diese oder jene oder gar das Entwickeln einer eigenen Technik ist zum Bestandteil der kreativen Arbeit des Komponisten geworden. Dabei erlauben verschiedene computergestützte Kompositionsstrategien dem Komponisten häufig radikal neue musikalische Formen.

[1] z.B. mit Gruppenkonzept von Stockhausen (*Gruppen* für Orchester) zu überwinden versucht.

Das oft eingesetzte prozedurale Programmierparadigma zwingt dem Komponisten jedoch eine ungewohnte sequenzielle Arbeits- und Denkweise auf: erzeuge die Daten des Parameters x erst mit dieser und bearbeite sie dann mit jener Funktion. Ein Komponist „mit Papier und Bleistift“ wird bei seiner Arbeit verschiedene musikalische Aspekte und deren Relationen berücksichtigen (etwa harmonische wie auch melodische Aspekte, instrumentationstechnische wie formale). Dieses Netz von Beziehungen verschiedenster Aspekte kann bei der Verwendung eines rein prozeduralen Algorithmus nur sehr schwer berücksichtigt werden.

Das Programmieren/Komponieren mit Constraints kann dafür eine Alternative darstellen: der Komponist definiert verschiedene Bedingungen, die das gesuchte Resultat erfüllen soll. Dies können harmonische oder melodische Bezüge sein — also Beziehungen zwischen dem gleichen Parameter verschiedener Töne — ebenso wie Bezüge zwischen beliebigen anderen Aspekten untereinander. Der Computer sucht eine Lösung, die den definierten Bedingungen entspricht. Ein solcher Ansatz ist dem traditionellen Denken eines Komponisten in Regeln viel vertrauter, beim Programmieren mit Constraints kann ein Komponist genau diese Regeln definieren.

2. Constraints

Dieser Abschnitt beschreibt zunächst die Idee und grundsätzliche Begriffe bei der Programmierung mit Constraints (Abschnitt 2.1). Da der Zweck des Constraintsprogramming leicht einsichtig ist, ist das Hauptaugenmerk der Literatur zu diesem Thema auf die effiziente Umsetzung gerichtet. Deshalb werden anschließend grundlegende Techniken zur Realisierung und Effizienzsteigerung bei der Constraints-Programmierung vorgestellt (Abschnitt 2.3). Diese wurden entweder für die Entwicklung von ARNO verwendet, oder es wird auf sie eingangen, wenn die offenen Fragen zu ARNO besprochen werden. Ein letzter Abschnitt stellt kurz verschiedene Programme vor, bei denen Constraints-Programmierung für musikalische Zwecke eingesetzt wurde und erläutert, worin sich die Zielsetzung für ARNO im Vergleich zu diesen Programmen unterscheidet (Abschnitt 2.4).

Algorithmen sind im Bemühen um möglichst leichte Verständlichkeit in Pseudocode mitgeteilt.

2.1. Idee

Bei *Constraints* (etwa: Bedingungen, Zwänge) handelt es sich um eine ganze Familie von Konzepten, die ermöglichen, daß der Programmierer nur Bedingungen definieren muß, die der Computer erfüllen soll:

> Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.
>
> Eugene C. Freuder [zitiert nach Barták, 1999, S. 1]

Solch eine Software-Technologie ist für kombinatorische Probleme und bei planerischen Aufgaben sehr hilfreich und wird deshalb z.B. in der Warenlager-Verwaltung oder bei der Erstellung von Fahrplänen eingesetzt. Aber auch die musikalische Komposition kann man unter kombinatorischen bzw. planerischen Gesichtspunkten sehen und deshalb wurden Constraints verschiedentlich zur algorithmischen Komposition verwendet.

Eine Constraint beschreibt eine logische Beziehung, die zwischen *Variablen* eingehalten werden soll. Jeder Variablen ist ein Wertebereich zugeordnet, ihre *Domain*. Eine Constraint untersagt Werte der Domain einer oder mehrer Variablen und repräsentiert so eine Teilinformation für diese Variablen. Eine Variable ist hier nicht ein Bezeichner (in der programmiertechnischen Wortbedeutung), sondern (in der mathematischen Bedeutung) wirklich eine Unbekannte. Um Verwechslungen zu vermeiden, werden solche Variablen im folgenden auch CSP-Variabeln[1] genannt.

[1] Constraints Satisfication Problem-Variable.

Eine Lösung besteht in einer Bindung für *jede* Variable, die innerhalb der Domain der jeweiligen Variablen liegt und alle diese Variable betreffenden Constraints erfüllt. Es kann eine, mehrere aber auch keine Lösung geben. Ziel ist es in der Regel, eine Lösung zu finden.

[Bartak, 1998, Introduction] zählt mehrere Eigenschaften von Constraints auf:

- Constraints können als *Teilinformationen* aufgefaßt werden, d.h. sie schränken die Domain ein, ohne einen bestimmten Wert zu determinieren.
- Constraints sind *deklarativ*: sie beschreiben eine Beziehung bzw. Bedingung, nicht aber die rechnerische Prozedur, diese zu erreichen.
- Constraints sind *additiv*: die Reihenfolge mehrerer Constraints ist nicht entscheidend, die Lösung muß lediglich alle erfüllen.
- Constraints sind *nicht direktional*: ein Constraint für die beiden Variablen x und y ist eine Teilinformation für x, wenn y durch andere Constraints determiniert ist und umgekehrt.

Ein musikalisches Beispiel soll den Constraintsbegriff verdeutlichen: es sollen die Tonhöhen zweier Noten (die Variablen) in der kleinen Oktave liegen (ihre Domain). Eine Constraint „die zweite Note sei höher als ihr Vorgänger“ ist eine Teilinformation, zumindest für die zweite Note. Eine Lösung bestünde in jedem Notenpaar innerhalb der kleinen Oktave, dessen zweite Note höher ist.

Es wurden Ansätze sowohl für Variablen mit endlicher als auch unendlicher Domain entwickelt, hier wird nur der Fall von Variablen mit endlicher Domain behandelt werden. Bei einem solchen Constraints Satisfication Problem (CSP) ist also gegeben:

- eine endliche Menge von Variablen
- jeder Variablen ist eine endliche Domain zugeordnet
- eine endliche Menge von Constraints

Zur Lösung eines CSP wurden verschiedene Ansätze vorgeschlagen, grundlegende Techniken werden im Abschnitt 2.3 vorgestellt.

2.2. Nicht-deterministische Programmierung

Dem Constraints-Paradigma nahe verwandt ist die Idee der nicht-deterministischen Programmierung. „A nondeterministic algorithm is one which relies on a certain sort of supernatural foresight“ [Graham, 1994, S. 286]. In einer nicht-deterministischen Programmierung-Umgebung können an beliebigen Stellen eines Programmes verschiedene Alternativen für nachfolgende Berechnungen eingeführt werden. Es muß nicht festgelegt werden, welche Alternative an dieser Stelle gewählt werden soll. Später im Programm werden die Resultate einzelner Alternativen untersagt. Das resultierende Programm entscheidet sich nur für erlaubte Alternativen.

In einer nicht-deterministischen Programmierung-Umgebung läßt sich Constraints-Programmierung einfach bewerkstelligen. Eine endliche Domain wird durch eine nicht-deterministische Auswahl aus den Werten der Domain deklariert. Constraints lassen sich als Verbote bestimmter Bindungen bzw. Bindungskombinationen ausdrücken.

Das schon erwähnte musikalische Beispiel läßt sich in einer nicht-deterministischen Umgebung z.B. wie folgt ausdrücken. Die Constraint „die zweite Note sei höher als ihr Vorgänger“ wird als Test formuliert. Den Tonhöhen zweier Noten werden nicht-deterministisch alle Tonhöhen der kleinen Oktave zugewiesen — ihre Domain. Auf die Noten wird der Test angewendet und alle Bindungen, die diesem Test nicht genügen, werden untersagt. Werden die Noten anschließend zurückgegeben, so genügen sie der Constraint.

Natürlich muß das „supernatural foresight“ eines nicht-deterministischen Algorithmus deterministisch simuliert werden. Dazu werden die Suchalgorithmen eingesetzt, die der folgende Abschnitt vorstellt.

2.3. Algorithmen

2.3.1. Systematische Suche

Algorithmen, die eine systematische Suche durchführen, binden jede CSP-Variable systematisch mit einzelnen Werten ihrer Domain. Dadurch finden diese Algorithmen mit Sicherheit eine Lösung, wenn eine solche existiert. Allerdings sind diese Verfahren wenig effizient. Die Darstellung folgt Bartak [1998] und Kumar [1992]

Generieren und Testen (Generate and Test)

Der *Generate-and-Test*-Algorithmus bindet alle Variabeln und testet anschließend, ob diese Lösung richtig ist. Stimmt die Lösung nicht, wird nach und nach systematisch jeder Wert der Domain jeder Variablen durchprobiert — der gesamte Suchraum entspricht also dem kartesischen Produkt aller Variablendomains. In der Regel wird nur nach der ersten Kombination aller Variablenbindungen gesucht, die allen Constraints genügt: diese stellt eine mögliche Lösung dar.

Nachteile: Dieser Algorithmus ist sehr ineffizient, da während der Suche die Variablen mit zuvielen falsche Werten gebunden werden. Die Suche geschieht völlig „blind“: vor der jeweils nächsten Bindung wird nicht ausgewertet, welche Variable den letzten Test nicht bestand; in Bezug auf den aufgetretenen Konflikt wird der Wert irgendeiner Variable geändert.

Zurücksetzten (Backtracking, BT)

Der Backtracking-Algorithmus vervollständigt nach und nach eine Teillösung zur gesuchten Lösung — im Gegensatz zu Generate-and-Test. Die aktuell untersuchte Variable $V_{current}$ wird mit einem Wert x ihrer Domain $D_{current}$ gebunden. Anschließend wird die Verträglichkeit dieser Bindung gegenüber den Contraints zu bis dahin erfolgten Bindungen getestet. Genügt die Bindung allen Constraints, wird eine nächste Variable $V_{current+1}$ in die Teillösung einbezogen. Diese jetzt aktuelle Variable wird mit einem Wert ihrer Domain gebunden, und diese Bindung wiederum

getestet. Versagt der Test für irgendeine Constraint, wird die aktuelle Variable nacheinander mit den übrigen Werten ihrer Domain gebunden und diese Bindungen getestet. Wird eine zulässige Teillösung gefunden, wird weiter zur folgenden Variablen fortgeschritten. Ist aber mit keinem Wert der Domain der aktuellen Variablen eine zulässige Teillösung zu finden, läuft der Algorithmus zurück (backtracks) zur Variablen vor der aktuellen Variablen und probiert deren Domain weiter durch. Sind alle Variablen mit einem zulässigen Wert gebunden, ist eine mögliche Lösung des CSP gefunden.[2]

Algorithmus 2.1: BT($V_{current}$)

```
procedur BT(V_current)
for each x ∈ D_current do
  SOLUTION_current ← x;
  if x fulfils all of its constraints then
    if V_current is last Variable then
      return SOLUTION
    else
      BT(V_current+1)
    end if
  end if
end for
```

BT führt also eine *Tiefensuche* (Depth-First-Search) durch, d.h. einen Algorithmus, der eine aktuell untersuchte Teillösung schnell so weit als möglich vervollständigt und auf alternative Suchpfade erst wenn nötig ausweicht.

Die Abb. 2.1[3] zeigt, wie das bekannte N-Damen-Problem von BT gelöst wird. Gesucht ist für n Damen auf einem Schachbrett mit der Seitenlänge n eine Stellung, in der keine der Damen eine andere angreift (in der Abb. ist $n = 4$). Jede Dame stellt eine CSP-Variable dar, deren Domain die n Felder ihrer jeweiligen Spalte umfaßt. Alle mißglückten Bindungen sind mit einem Kreuz markiert. Wurden alle Werte der Domain erfolglos durchprobiert, findet ein BT statt.

Durch das Zurücklaufen bei Verletzung irgendeiner Constraint wird der Suchraum des Algorithmus im Vergleich zum Generate-and-Test für die meisten Probleme erheblich reduziert: eine Suche durch BT ist meist deutlich effizienter als mit Generate-and-Test. Dennoch kann auch der BT-Algorithmus alle Lösungen eines mit Constraints formulierten Problems finden — keine der übersprungenen Teillösungen hätte zu einer vollständigen Lösung geführt.

Nachteile: Die Suche ist bei nichttrivialen Problemen trotzdem sehr aufwendig, die zum Finden einer Lösung benötigte Zeit steigt auch beim BT mit zunehmender Komplexität des Problems exponentiell an. Bartak [1998, Systematic Search Algorithms] faßt die Probleme übersichtlich zusammen:

- Auch BT sucht blind: es treten wiederholt Fehler mit derselben Ursache auf (*Trashing*), da BT nicht berücksichtigt, welche Variablenbindungs-Kombination eine Contraint verletzt, also welche Variable vor der aktuellen Variable die Ursache des Fehlers ist.

[2] Der mitgeteilte Algorithmus ist — übertragen in Pseudocode — zitiert nach Kondrak [1994, S. 9]. Der Algorithmus findet allerdings nur die erste Lösung.

[3] Abb. nach Bartak [1998]

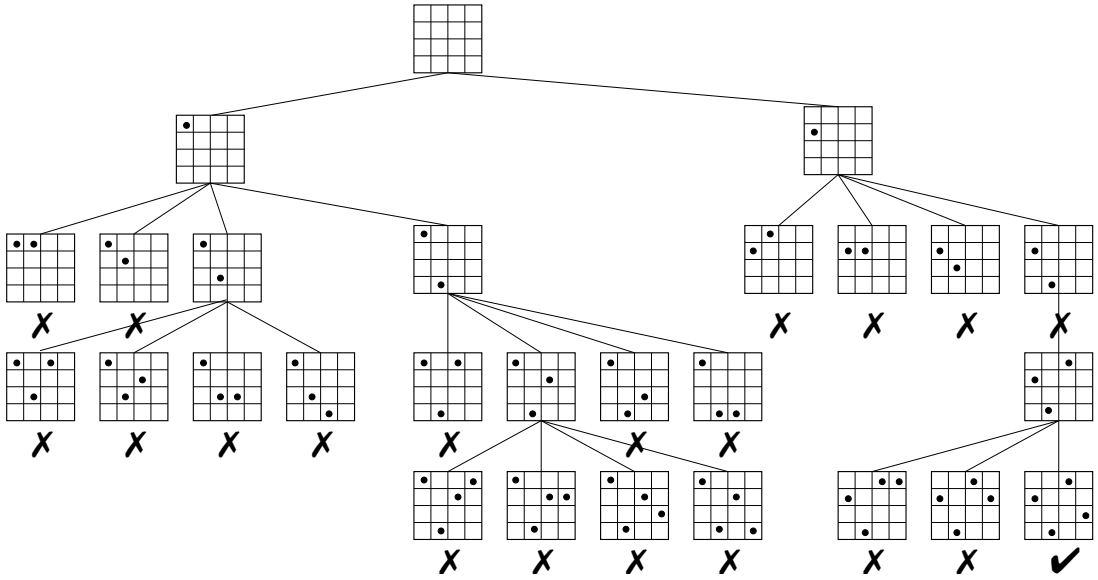

Abbildung 2.1.: Das 4-Damen-Problem mit Backtracking gelöst

- Selbst wenn die fehlerverursachende Variablenbindung vom Algorithmus erkannt und direkt zu dieser Variablen zurückgesetzt wird, tritt dieser Fehler erneut auf, wenn im späteren Verlauf der Suche *vor* diesen zurückgegangen werden muß (*überflüssige Arbeit,* redundant Work).
- BT entdeckt einen Konflikt immer zu spät, d.h. erst wenn er nach Bindung aller betroffenen Variablen tatsächlich eintritt.

Erweiterung des BT

Um BT effizienter zu gestalten wurden verschiedene Erweiterungen entwickelt, die den eben benannten Punkten begegnen sollen.

Durch *Intelligentes Zurücksetzen* (Intelligent Backtracking, Backjumping, BJ) läßt sich das Trashing[4] vermeiden, indem der Algorithmus zu der Variablen zurückspringt, die den Konflikt verursacht. Der Algorithmus muß dazu ermitteln können, welche Variable eine Constraint der aktuellen Variable verletzt. Wenn in der Domain der aktuellen Variablen kein Wert allen Constraints für diese Variable genügt, läuft der Algorithmus nicht zur vorigen Variabeln zurück (chronologisches BT), sondern springt direkt zur nächsten durch eine verletzte Constraint „betroffenen“ Variablen.

Das *Abhängigkeitsgeleitete Rücksetzen* (Dependency-Directed Backtracking), eingesetzt in *Begründungsverwaltungssystemen* (Truth-Maintenance Systems), vermeidet sowohl Trashing als auch überflüssige Arbeit, indem der Algorithmus nach fehlgeschlagenen Tests bestimmter Variablenbindungen diese Bindungskombination als nicht möglich protokolliert. Dieser Ansatz aber kann im ungünstigen Fall aufwendiger sein als chronologisches BT.

[4] Siehe oben.

Um mögliche Konflikte schon im voraus reduzieren zu können, wurde *Forward Checking* entwickelt.[5]

Für alle diese Erweiterungen des chronologischen BT gilt: durch sie wird die Anzahl aller während der Lösungsfindung besuchten Variablen reduziert, indem bei jeder einzelnen Variablen ein rechnerischen Mehraufwand getrieben wird. Durch diesen Mehraufwand kann — abhängig vom behandelten CSP — ein solcher Algorithmus durchaus aufwendiger als einfacher BT sein. Zum anderen muß die Problemformulierung für die verschiedenen Erweiterungen des BT angepaßt werden. Auch sind die verfeinerten Algorithmen schwerer zu verstehen.

2.3.2. Konsistenztechniken

Zu den Konsistenztechniken gehören verschiedene Algorithmen, die nicht — wie die eben betrachteten Suchalgorithmen — Variablenbindungen „durchprobieren“: ein solcher Algorithmus soll in der Domain von Variabeln eines CSP Werte herausfinden, die nicht zu einer Lösung gehören können und also aus der Domain entfernt werden können.

Die Beschreibung der Algorithmen verwendet — zusätzlich zu und neben den bereits eingeführten Begriffen der Variable, ihrer Domain und der die Variable näher beschreibenden Constraint — Begriffe der Graphentheorie. Der Begriff *Knoten* (Node) wird synonym zu einer CSP-Variablen verwendet. Eine Constraints zwischen zwei Knoten wird als *Kante* (Arc) bezeichnet. Eine Menge von Knoten, durch Kanten miteinander verbunden, bilden einen Constraintsgraphen bzw. ein Constraintsnetz.

Eine *einstellige* (unary) Constraint ist eine Constraint, die sich auf nur eine Variable bezieht, formal repräsentiert durch eine Kante der in demselben Knoten beginnt und endet. Eine *binäre* Constraint dagegen beschreibt die Beziehung genau zweier Variablen zueinander. In der Graphenrepräsentation sind dazu zwei Knoten durch eine Kante miteinander verbunden.

Natürlich sind auch *mehrstellige* (n-ary) Constraints praktisch sinnvoll. Die meisten Algorithmen zum Lösen eines Constraintnetzes sind aber auf einstellige und binäre Constraints beschränkt. Jedoch läßt sich jedes CSP mit mehrstelligen Constraints mit Hilfe von zusätzlichen Hilfsvariablen in ein äquivalentes CSP mit binären Constraints umformen [siehe Bartak, 1998, Binarization of Constraints].

Knoten-Konsistenz (Node Consistency, NC)

Diese Technik ist die einfachste. Die Domain einer Variable V ist dann knoten-konsistent, wenn für jeden Wert ihrer Domain D_V alle einstelligen Constraints erfüllt sind. Der Algorithmus [zitiert nach Bartak, 1998, Consistency Techniques] muß also bei allen Variablen alle die Werte x aus der Domain entfernen, die nicht den einstelligen Constraints genügen.

Ein musikalisches Beispiel: die Tonhöhe einer Note (Variable), deren Domain alle chromatischen Stufen vom kleinen c bis h’ umfaßt, soll ein leitereigener Ton von Es-Dur sein und zum D^7 dieser Tonart gehören (zwei einstellige Constraints). Nach einer Reduzierung ist die Rest-Domain dieser Note ist knoten-konsistent, wenn die Rest-Domain nur noch die Töne b, d, f und as aus den beiden Oktaven der ursprünglichen Domain umfaßt — die ursprüngliche Domain wurde damit auf ein Drittel reduziert.

[5] Siehe Abschnitt 2.3.3.

Algorithmus 2.2: NC

```
procedur NC
for each V ∈ nodes(G) do
  for each x ∈ D_V do
    if any unary constraint on V is inconsistent with x then
      delete x from D
    end if
  end for
end for
```

Kanten-Konsistenz (Arc Consistency, AC)

Eine Kante(V_i, V_j) ist kanten-konsistent, wenn es für jeden Wert x in der Domain D_i der Variablen V_i mindestens einen Wert y in der Domain D_j von V_j gibt, der alle binären Constraints zwischen V_i und V_j erfüllt. Alle Werte der Domain D_i ohne ein solches y können aus D_i entfernt werden: sie können zu keiner Lösung gehören. Damit ist die Technik der Kanten-Konsistenz direktional, wenn eine Kante(V_i, V_j) kanten-konsistent ist, gilt dies nicht automatisch auch für die Kante(V_j, V_i).

Ein musikalisches Beispiel: im traditionellen Kontrapunkt erlauben Stimmführungsregeln nur bestimmte Intervalle zwischen den Tonhöhen einer Stimme (eine Constraint bzw. eine Kante). Enthält nun die Domain (D_i) eines Vorgängertons (V_i) eine Tonhöhe (x), für die es in der Domain (D_j) des Nachfolgetons (V_j) mit keinem der erlaubten Intervalle eine mögliche Nachfolgetonhöhe gibt (d.h. es gibt kein y), so kann diese Tonhöhe (x) aus der Domain des Vorgängertons entfernt werden — sie kann kein Bestandteil einer Lösung sein.

Der Algorithmus REVISE [Mackworth, 1977][6] testet eine Kante auf Konsistenz, entfernt nichtkonsistente Werte und gibt zurück, ob die Domain geändert wurde.[7]

Algorithmus 2.3: REVISE(V_i, V_j)

```
procedur REVISE(V_i, V_j)
DELETE ← false
for each x ∈ D_i do
  if there is no such y ∈ D_j
  such that (x, y) is consistent then
    delete x from D_i;
    DELETE ← true
  end if
end for
return DELETE
```

Um zu erreichen, daß ein CSP vollständig kanten-konsistent ist, genügt es nicht, nacheinander jeden Knoten mit REVISE zu kontrollieren — wurde bei der Konsistenzkontrolle der Kante(V_j, V_k) die Domain D_j reduziert, so müssen alle zuvor schon konsistenten Kanten(V_i, V_j) erneut kontrolliert werden.

[6] Zitiert nach Kumar [1992, S. 5]

[7] Wenn die Domain geändert wurde, wird *true* zurückgegeben, andernfalls *false*.

Der Algorithmus AC-1 [Mackworth, 1977][8] leistet dies, indem die Ausgabe von REVISE ausgewertet wird.

Algorithmus 2.4: AC-1

procedur AC-1
$Q \leftarrow \{(V_i, V_j) \in arcs(G), i \neq j\}$;
repeat
  $CHANGE \leftarrow false$;
  for each $(V_i, V_j) \in Q$ **do**
    $CHANGE \leftarrow (REVISE(V_i, V_j)$ or $CHANGE)$
  end for
until not$(CHANGE)$

AC-1 ist nicht sehr effizient, da bei jeglicher erfolgreichen Revision alle Knoten erneut kontrolliert werden, unabhängig davon, ob sie von der Änderung überhaupt betroffen sein können. So müssen nach einer Reduktion der Domain der Variablen V_k nur alle Kanten(V_i, V_k) erneut revidiert werden. Jedoch muß — wurde V_k nach Revision der Kante(V_k, V_m) reduziert — nicht auch die Kante(V_m, V_k) erneut untersucht werden, denn ein aus der Domain von V_k entferntes Element hat ja keinen Wert der Domain von V_m unterstützt. Eine derart verfeinerte Fassung stellt der Algorithmus AC-3 [Mackworth, 1977][9] dar:

Algorithmus 2.5: AC-3

procedur AC-3
$Q \leftarrow \{(V_i, V_j) \in arcs(G), i \neq j\}$;
while Q not empty **do**
  select and delete any $arc(V_k, V_m)$ from Q;
  if $REVISE(V_k, V_m)$ **then**
    $Q \leftarrow Q \cup \{(V_i, V_k)$ such that $(V_i, V_k) \in arcs(G), i \neq k, i \neq m\}$
  end if
end while

Verschiedene andere Verfeinerungen wurden vorgeschlagen, doch AC-3 gehört zu den häufiger eingesetzten Algorithmen zum Erreichen der Kanten-Konsistenz.

Die Technik der Kanten-Konsistenz kann viele inkonsistente Werte aus den Domains eines CSP entfernen. Eine (und die einzige) Lösung würde mit dieser Technik aber erst gefunden werden, wenn die Domain jeder Variablen nur noch einen Wert enthält. Andernfalls muß noch eine systematische Suche in dem (allerding u.U. erheblich reduzierten) Suchraum vorgenommen werden, um zu einer Lösung zu gelangen.

Die Technik der Kanten-Konsistenz kann nicht erkennen, ob ein CSP überhaupt eine Lösung hat, wenn die Inkonsistenz erst zwischen mehr als 2 Knoten auftritt. Ebenso stellen geschlossene Kreise von Kanten und Knoten ein Problem dar. Es seien z.B. in einem CSP drei Knoten, jedes mit der Domain $\{1, 2\}$, jeweils mit der Kante Nachbar $\neq$ Nachbar miteinander vebunden (Abb. 2.2). Alle Kanten sind konsistent, aber es gibt keine Lösung.[10]

[8] Zitiert nach Kumar [1992, S. 5f].
[9] Zitiert nach Kumar [1992, S. 6].
[10] Abb. nach Bartak [1998].

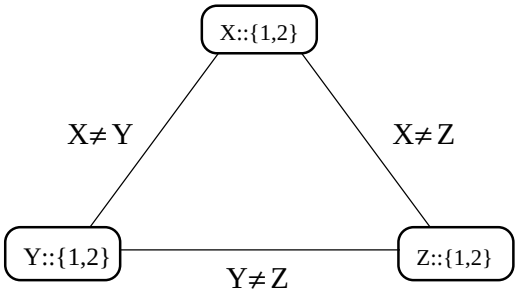

Abbildung 2.2.: Ein kanten-konsistentes CSP ohne Lösung

Dieser Grenze der Kanten-Konsistenz läßt sich mit einer Konsistenz-Technik höheren Grades, die mehr Knoten einbezieht, begegnen (K-Konsistenz).[11] Es sind auch Algorithmen entwickelt worden, die geschlossene Kreise gesondert behandeln (Cycle-Cutset), denn ein CSP ohne geschlossene Kreise kann durch Kanten-Konsistenz-Kontrolle vollständig konsistent gemacht werden. Wichtig ist dafür jedoch die Reihenfolge der Knoten während der Suche [siehe Kumar, 1992, S. 8ff].

2.3.3. Forward Checking

Die Technik des Forward-Checking stellt eine reduzierte Form einer Kanten-Konsistenzkontrolle dar: es wird jeweils nur eine Konsistenz zwischen wenigen Knoten erzielt. Anstatt wie beim chronologischen Backtracking die Konsistenz der schon gebundenen Knoten zur ebenfalls gebundenen aktuellen Knoten nachträglich zu testen, wird eine Konsistenzkontrolle der noch nicht gebundenen „zukünftigen“ Knoten, von denen Kanten zum aktuellen Knoten führen, durchgeführt. Die Domains dieser zukünftigen Knoten werden (zeitweise) entsprechend reduziert. Ist die Domain eines zukünftigen Knotens danach leer, findet ebenfalls ein BT statt. Durch diesen (im Vergleich zum chronologischen BT höheren) Aufwand bei der Bindung jedes Knotens wird jedoch der Suchbaum insgesamt kleiner [siehe Bartak, 1998, Constraints Propagation].

Die Abb. 2.3[12] zeigt, wie das N-Damen-Problem mit Forward Checking gelöst wird. Durchkreuzte Felder markieren Werte, die aus der Domain der Dame der jeweiligen Spalte entfernt wurden. Der Suchbaum ist deutlich kürzer als beim Lösen desselben Problems mit BT (siehe Abb. 2.1), da weniger häufig falsche Bindungen vorgenommen werden (wieder markiert durch Kreuze).

2.3.4. Anordnung von Variablen und Werten der Domain

Ein Suchalgorithmus arbeitet nacheinander alle CSP-Variablen ab. Auch die Werte der Domain einer Variablen werden nacheinander eingesetzt. Dabei kann die Reihenfolge der Variablen und Werte für die Effizienz der Suche entscheidend sein.

[11] Knoten-Konsistenz und Kanten-Konsistenz sind Teilmengen der K-Konsistenz und können auch mit 1-Konsistenz bzw. 2-Konsistenz bezeichnet werden.

[12] Abb. nach Bartak [1998]

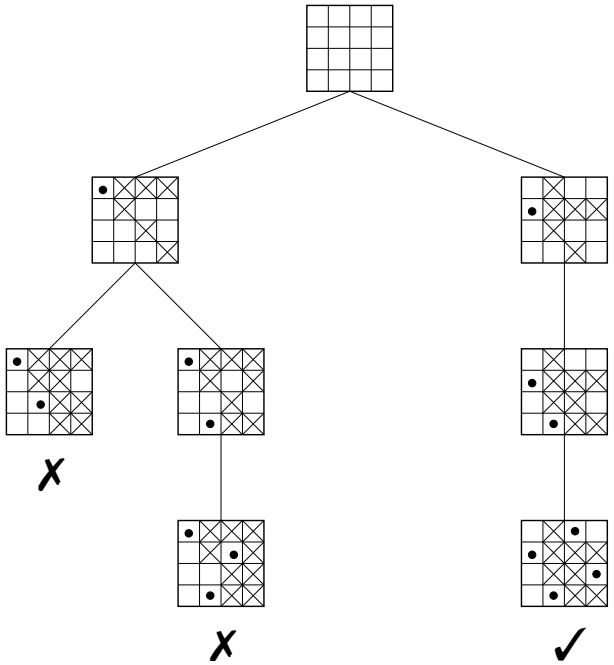

Abbildung 2.3.: Das 4-Damen-Problem, gelöst mit Forward Checking

Weder die Reihenfolge der Variablen noch die Reihenfolge der Domain-Werte ist allerdings von Bedeutung, wenn alle Lösungen gesucht werden oder wenn es keine Lösung gibt, denn in diesen Fällen ist eine erschöpfende Suche nötig, die alle Variablen- und Wertekombinationen durchprobiert.

Anordnung der Variablen

Die Anordnung der CSP-Variablen für eine Suche kann entweder statisch oder dynamisch sein. Bei einer statischen Anordnung ist die Reihenfolge der Variablen vor dem Start der Suche festgelegt und ändert sich nicht. Bei einer dynamische Anordnung ist die Wahl der nächsten Variablen jeweils abhängig von aktuellen Stand der Suche.

Ein Algorithmus mit dynamischer Anordnung der Variablen braucht für die Entscheidung, welche Variable als nächste zu besuchen ist, zusätzliche Informationen. Diese können z.B. auf der Größe der Domain der noch ungebundenen Variablen oder der Anzahl der sie betreffenden Constraints beruhen. Um ein Constraints-Problem möglichst schnell zu lösen, wurden verschiedene Heuristiken für diese Entscheidung entwickelt.

Häufig eingesetzt wird die Heuristik *First-Fail*: „Versuche zuerst die schwierigen Fälle!“ Solche schwierigen Fälle sind Variablen mit relativ kleiner Domain bzw. Variablen, auf die sich relativ viele Constraints beziehen.[13] Hier ist ein Fehlschlag möglichst früh wünschenswert bzw. eine Bindung später in der Suche schwieriger, als bei Variabeln mit größerer Domain oder weniger Constraints.

[13] Es wird dafür angenommen, daß alle Domain-Werte gleich warscheinlich bzw. alle Constraints gleich schwer zu erfüllen sind.

Die für eine dynamische Anordnung der Variablen benötigten Informationen stehen z.B. während einer Suche mit chronologischen Backtracking nicht zur Verfügung. Aber auch für die statische Anordnung der Variablen sind Heuristiken vorgeschlagen worden, z.B. „Sortiere die Variablen so, daß Variablen mit einer großen Anzahl von Constraints zu Variablen *vor* ihnen möglichst früh besucht werden!“

Anordnung der Werte in einer Domain

Ist die Entscheidung gefallen, welche Variable als nächstes gebunden werden soll, so muß entschieden werden, mit welchem Wert der Domain dieser Variablen die Bindung geschieht. Idealerweise wird eine Variable sofort mit einem Wert gebunden, der zur Lösung gehört, denn dann ist kein Backtracking erforderlich. Für dieses *Succeed -First* genannte Prinzip werden die Werte der Domain entsprechend einer Heuristik nach Erfolgswahrscheinlichkeit geordnet: „vorn“ in der Domain befindet sich der am wahrscheinlichsten erfolgreiche Wert, mit dem zuerst eine Bindung vorgenommen wird.

Eine solche Heuristik kann z.B. ein den Konsistenzkontrollen ähnlicher Algorithmus sein, der zählt, wieviele Werte in den Domains der noch ungebundenen Variablen der jeweilige Wert zuläßt. Je mehr Möglichkeiten ein Wert zuläßt, desto wahrscheinlicher kann er zu einer Lösung gehören. Eine andere Heuristik mißt, wieviel Prozent der Domains noch ungebundener Variablen der jeweilige Wert *nicht* zuläßt. Je weniger die Domains noch ungebundener Variablen durch einen Wert reduziert werden, desto warscheinlicher ist er Bestandteil einer Lösung.

Durch die Reihenfolge der Domain-Werte läßt sich u.U. auch die Güte einer Lösung beeinflussen, d.h. die Anordnung der Variablen beeinflußt nicht nur die Effizienz der Suche, sondern auch das Ergebnis. Je weiter vorn in der Domain ein Wert steht, desto wahrscheinlicher wird die Variable mit diesem Wert gebunden. Es sollten deshalb die Werte vorn stehen, die zu einer besseren Lösung gehören können. Die Güte einer Lösung läßt sich sehr schwer allgemein, unabhängig vom jeweils zu lösenden Constraints-Problem, definieren. Es kann z.B. bei einem Problem mit mehreren Variablen mit je gleicher Domain eine gleichförmige Lösung, bei der viele oder sogar jede Variable mit einem gleichen Wert gebunden ist, unerwünscht sein. In diesem Fall besteht eine bessere Lösung in weniger gleichförmigen Bindungen, die durch eine zufällige Reihenfolge der Domain jeder Variablen zu erreichen ist.

Es ist sowohl für die Erhöhung der Effizienz der Suche wie auch für die Verbesserung der Güte einer Lösung schwer, allgemeine Heuristiken zu formulieren. Die Brauchbarkeit einer Heuristik ist vom konkret zu lösenden Constraints-Problem abhängig.

2.4. Musikalische Projekte, die Constraints einsetzen

Es sind verschiedene Projekte entwickelt worden, die Constraints-Programmierung für musikalische Zwecke einsetzen. Das Generate-and-Test-Paradigma wurde bereits 1956 für das Komponieren der *Illiac Suite for String Quartet* — der ersten unter Einsatz eines Computers komponierten Komposition überhaupt — eingesetzt [Hiller and Isaacson, 1993]. Verschiedene spätere musikalische Anwendungen von Generate-and-Test werden von Roads [1996, S. 892 ff] benannt.

Im folgenden werden kurz mehrere Programme vorgestellt, bei denen Constraints mit effizienteren Algorithmen als Generate-and-Test, insbesondere Backtracking, eingesetzt wurden, um

musikalische Ergebnisse zu erzielen. Ein anschließender Abschnitt erläutert ihr Ungenügen für die beabsichtigten kompositorischen Ziele, die die Entwicklung einer weiteren Umgebung nötig machte.

2.4.1. Chorale

Ebcioğlu [1992, 1993] entwickelte mit CHORALE ein System, das Choräle im Stile Johann Sebastian Bachs vierstimmig harmonisieren soll. Dazu wurde BSL,[14] eine neue und effiziente Logikprogrammiersprache, entwickelt und in dieser etwa 350 Kompositionsregeln formuliert. Diese umfassen Gesichtspunkte wie Stimmführung, die besondere Behandlung der Außenstimmen und harmonische Regeln (Akkordskelett, stilgemäße Modulation und Kadenzierung). Die in CHORALE eingesetzten Constraints sind aus verschiedenen Lehrwerken der Musiktheorie abgeleitet. Sie werden ergänzt um Heuristiken für die Anordnung der Werte in einer Domain.

Durch eine große stilistische Beschränkung auf den vierstimmigen Choralsatz, ein bevorzugtes musiktheoretisches Forschungsobjekt, für den im Rahmen des Projektes dennoch eine sehr große Anzahl von Constraints formuliert wurden, erreicht das Programm nach Angaben seines Autors eine Kompetenz, die sich der eines talentierten Musikstudenten nährt [vg. Ebcioğlu, 1993, S. 384].

2.4.2. Programmierte Jazzimprovisation

Der Fokus der von Levitt [1993] am MIT entwickelten Software ist ebenfalls auf die Erzeugung eines bestimmten Stils gerichtet. Für das Programm wurde eine Sprache zur Beschreibung musikalischer Stile entwickelt und mit dieser stilistische Eigenheiten des klassischen Jazz bzw. Ragtime beschrieben. Naturgemäß ist dabei großes Gewicht auf die Erzeugung von Akkordfolgen, ihres (pianistischen) Arrangements und der Generierung von in dieses harmonische Schema passenden Melodien gelegt worden, wobei das System keinerlei Zufallsoperatoren einsetzt.

2.4.3. Situation

SITUATION, von Camilo Rueda am IRCAM, Paris, entwickelt [Rueda and Bonnet, 1996], ist in COMMON LISP geschrieben. Das Programm umfaßt eine Suchmaschine mit Back Tracking, einen Pattern Matching-Mechanismus[15] und eine Menge vordefinierter Constraints, denen Argumente übergeben werden können. Ursprünglich wurde SITUATION zur Generierung von Akkordfolgen entwickelt, inzwischen ist es verschiedentlich erweitert worden. SITUATION ist eine Erweiterungbibliothek der ebenfalls am IRCAM entwickelten Kompositionsumgebung PATCHWORK (PW).[16] Dadurch können die Ergebnisse mit den Mitteln von PW visualisiert, ausgegeben und auch weiter bearbeitet werden.

[14] Die *Backtracking Specification Language* ist eine Sprache, die Nichtdeterminismus mit intelligentem Backtracking (Back Jumping) realisiert, über PASCAL-ähnlichen Datentypen und einer LISP-artigen Syntax verfügt. Mit einem LISP-Programm werden in BSL geschriebene Programme zu C kompiliert.

[15] Pattern-Matching ist ein Mechanismus, durch den mit einem allgemeinen Muster eine bestimmte Instanz (oder ein anderes Muster) ausgedrückt werden kann, auf die das allgemeine Muster paßt. Üblich ist z.B. daß Wildcards eingesetzt werden können, die wie Joker für beliebige Zeichen (häufig das Zeichen `?`) oder Zeichenfolgen (häufig das Zeichen `*`) stehen.

[16] Bei PATCHWORK werden einzelne Boxen, die jeweils LISP-Funktionen repräsentieren, graphisch durch „Patchchords“ verbunden — daher auch der Name.

2.4.4. PatchWork Constraints (PWConstraints)

Auch PatchWork Constraints [Laurson, 1996] ist (wie Situation, siehe oben) eine Erweiterungsbibliothek der Kompositionsumgebung PatchWork des IRCAM. PWConstraints erlaubt eine sehr freie Deklaration der Tonhöhen einer zumindest rhythmisch vordefinierten musikalischen Partitur. Es ist ein komplexes Partiturmodell implementiert, das ein Referenzieren der übergebenen Rhythmuspartitur nach Stimmen, Akkorden und sogar Taktschwerpunkten ermöglicht. Das Suchen der Lösung geschieht intern mittels Back Tracking, allerdings sind mit besonderen Deklarationen auch Forward Checking, Heuristiken, die die Suchreihenfolge der CSP-Variablen ändern, und gewichtete Constraints[17] möglich. Weil PWConstraints eine Bibliothek von PW ist, können Ergebnisse von PWConstraints mit den Mitteln von PW dargestellt,[18] ausgegeben[19] und auch weiter bearbeitet werden.[20] Die Formulierung der Deklarationen allerdings erfolgt — ungewöhnlich für PW, in der die Programmierung sonst nur graphisch erfolgt — in Textform.

Zur Deklaration von Constraints zwischen Noten wird in PWConstraints ein Pattern Matching-Mechanismus eingesetzt. Eine *Regel* (wie eine Constraint in PWConstraints genannt wird) setzt sich zusammen aus einem Pattern Matching-Teil, der bei Zutreffen den anschließenden Lisp Code-Teil auslöst. Mit dem Pattern Matching-Mechanismus in PWConstraints ist es recht kompakt möglich, auf andere Töne innerhalb derselben Stimme zu referenzieren, um z.B. die Tonhöhe des aktuellen Ton von dem Vorgängerton abhängig zu machen. Ein Test „Gibt es einen Vorgängerton?“ muß vom Nutzer nicht extra definiert werden sondern ist im Pattern Matching-Mechanismus implizit. Die verwendete Syntax des Pattern Matching Part allerdings ist sehr kompakt, die Bezeichner sind sehr kurz und nur nach einer gewissen Lernphase intuitiv verständlich.

2.4.5. Warum zusätzlich Arno

Die in den vorangegangenen Abschnitten vorgestellten Programme stellen verschiedene musikalische Anwendungen der Constraints-Programmierung dar. Sie alle lösen die in der Einleitung[21] beschriebenen Probleme, die sich aus der Anwendung des prozeduralen Programmierparadigmas für die computergestützte Komposition ergeben. Auch handelt es sich um z.T. sehr ausgereifte Programme. Warum sollte noch ein weiteres, ähnliches Programm entwickelt werden?

Arno wurde entwickelt, um einem Komponisten ein flexibles Werkzeug in die Hand zu geben. Bei den meisten der vorgestellten Programme spielt dagegen die musikalische Stilkopie eine große Rolle. Diese Einschränkung fällt allerdings wenig ins Gewicht. Wie in der Einleitung[22] ausgeführt sind gerade Stilkopien gut geeignet, die Leistungsfähigkeit eines Programms zu demonstrieren. Das gleiche Programm läßt sich ebenso anders einsetzen. Auch die Anwendung von Arno wird u.a. deshalb mit einer traditionellen Satztechnik demonstriert.[23]

[17] Sind nicht alle Constraints vollständig zu erfüllen, wird der Constraint mit der höheren Gewichtung gegenüber der mit einer geringeren Gewichtung der Vorzug gegeben.
[18] Eine durch PWConstraints erzeugte Partitur wird in einem PW-Editor in traditioneller Notation visualisiert.
[19] In der Regel geschieht die Ausgabe im MIDI-Format.
[20] Die Bearbeitung ist interaktiv in einem graphischen Editor von PW oder durch andere in PW geschriebene Programme möglich.
[21] Siehe S. 6.
[22] Siehe S. 5.
[23] Siehe Abschnitt 4.4.

Schwerer dagegen fällt ins Gewicht, daß nicht jedes der vorgestellten Programme in eine allgemeine, umfangreiche Umgebung zur musikalischen Komposition integriert ist. Nur SITUATION und PWCONSTRAINTS sind Bibliotheken der Kompositionsumgebung PATCHWORK. Dadurch können die Resultate dieser Programme komfortabel mit anderen PW-Programmen oder manuell in einem PW-Editor weiter bearbeitet und in verschiedenen Formaten ausgegeben werden. Darüber hinaus jedoch sind die Programme von Ebcioğlu und Levitt offenbar gar nicht allgemein verfügbar, sondern wurden nur in Publikationen vorgestellt.

Die somit verbleibenden Programme SITUATION und PWCONSTRAINTS sind verschiedentlich eingeschränkt. Bei der SITUATION-Version, die erhältlich war, als der Autor mit der Arbeit an ARNO begann, konnte ein Komponist für bereits programmierte Constraints nur Parameter frei einsetzten. Eine derart große Einschränkung machte das Programm für die kompositorische Arbeit des Autors unbrauchbar.

Deutlich freier gestaltet sich der Umgang mit PWCONSTRAINTS. Dieses Programm hat deshalb in verschiedener Hinsicht als Vorbild bei der Entwicklung von ARNO gedient. Doch auch die Möglichkeiten mit PWCONSTRAINTS sind deutlich eingeschränkt. Es ist vor allem nur ein einziger musikalischer Parameter, die Tonhöhe, deklarierbar. Anderen Parameter — insbesondere die Zeitstruktur — müssen zuvor erstellt werden und lassen sich von PWCONSTRAINTS nicht ändern. Aber sogar die Deklaration der Tonhöhen ist eingeschränkt: deklarierbar sind nur MIDI-Tonhöhen, d.h. zwölf Stufen pro Oktave. Mit PWCONSTRAINTS ist eine (vom Autor angestrebte) mikrotonale Harmonik nur eingeschränkt und umständlich realisierbar.

Diese Einschränkungen von PWCONSTRAINTS (nur die Tonhöhe und diese nur in zwölf Stufen pro Oktave deklarierbar) aufzuheben war eine der entscheidenden Motivationen für die Entwicklung von ARNO: Ziel war ein Werkzeug zur computergestützten Komposition, das die Formulierung möglichst beliebiger kompositorische Regeln erlaubt. Es war nicht möglich, PWCONSTRAINTS entsprechend zu erweitern. Eine Anfrage nach dem Quelltext von PWCONSTRAINTS wurden vom IRCAM abgewiesen. Die Kompositionsumgebung COMMON MUSIC (siehe Abschnitt 3.2) ist im Gegensatz zu PATCHWORK offene, freie Software, für die keine Erweiterung zur Constraints-Programmierung existierte. Deshalb wurde diese Umgebung zur Entwicklung von ARNO gewählt.

3. Verwendete Umgebung zur Verwirklichung von Arno

Das folgende Kapitel stellt die für Arno verwendete Programmiersprache Common Lisp und zwei verwendete Erweiterungen vor: die Kompositionsumgebung Common Music (CM) und Screamer, ein Werkzeug zur Constraints-Programmierung. Angeschlossen ist eine Begründung für die Wahl dieser Umgebungen und Bemerkungen zu ihrem Zusammenspiel in Arno.

3.1. Common Lisp

Die Programmiersprache Common Lisp [Steele, 1990],[1] im weiteren auch Lisp genannt, ist eine mächtige, standardisierte und sehr reiche Sprache. Sie hat eine im Prinzip sehr einfache Syntax. Sie ist wenig maschinennah, es ist z.B. keine explizite Speicherverwaltung oder Deklaration von Variablentypen notwendig.

In der Syntax wird kein prinzipieller Unterschied zwischen Programmen und Daten gemacht. Dadurch können beispielsweise Funktionen auch Argumente von Funktionen sein oder Programme als Ergebnis Programme zurückgeben. Lisp läßt sich einfach erweitern, es lassen sich auch gut eigene Sprachen in Lisp definieren:

> Lisp is a programmable programming language.
>
> John Foderaro [zitiert nach Graham, 1994, S. v]

Lisp ist traditionell eine interpretierende Sprache, die Programmentwicklung erfolgt interaktiv — jedes Programm (z.B. eine Hilfsfunktion) kann sogleich getestet werden. Als objektorientierte Erweiterung von Common Lisp hat sich Clos [Keene, 1989] durchgesetzt, das in CM und damit auch in Arno Anwendung findet.

Im Anhang A.1 findet sich eine kurze Einführung in die Programmierung von Makros in Lisp. Die Makroprogrammierung in Lisp eröffnet deutlich weitergehende Freiheiten als die Makroprogrammierung in anderen Sprachen, wovon zur Entwicklung des Kernes von Arno Gebrauch gemacht wurde. Der Anhang A.2 führt in die Idee und Begriffe der objektorientierten Programmierung mit CLOS ein. Im Anhang A.3 wird eine Möglichkeit gezeigt, Nicht-Determinismus in Lisp zu implementieren.

[1] Als Referenz gilt Steele [1990], leichter lesbar dagegen ist [HyperSpec]. Eine gute Einführung in Common Lisp in deutscher Sprache bietet Graham [1995].

3.2. Common Music (CM)

Common Music [Taube, a,b] ist eine Umgebung zur computergestützten musikalischen Komposition. CM ist in Common Lisp geschrieben und erweitert Lisp um flexible Sprachmittel zum Erstellen musikalischer Strukturen, die mit Hilfe einer internen, hoch strukturierte Partiturrepräsentation dargestellt werden. CM bietet mit Stella [Taube, 1994, c] einen textbasierten Editor zum Bearbeiten und Analysieren der internen Partitur.[2]

Die interne Partiturrepräsentation kann CM in eine Vielzahl von Ausgabe-Formaten transformieren.[3] CM definiert eine umfangreiche Bibliothek von Kompositionswerkzeugen und stellt eine Anwendungs-Schnittstelle (Application Program Interface, API) zur Verfügung, mit deren Hilfe der Komponist das System gut erweitern kann. Da CM in Common Lisp geschrieben ist, läuft es auf einer Vielzahl von Plattformen.[4]

CM unterstützt das Konzept der sogenannten Item-Streams besonders: verschiedene Pattern (wie z.B. zyklische Wiederholung, Palindrom, zufällige Anordnung gegebener Werte, Markowketten und auch selbst definierte Pattern) können beliebig miteinander kombiniert und ineinander verschachtelt werden. Die Werte der resultierenden Streams können beliebigen musikalischen Parametern zugewiesen werden.[5] Die Item-Streams werden für Arno nicht genutzt: Ziel der Entwicklung von Arno war gerade, CM um eine andere kompositorische Strategie, die Constraints-Programmierung einsetzt, zu erweitern.

Entwickelt wurde CM von Heinrich Taube, begonnen 1989 am Stanford University Center for Computer Research in Music and Acoustics (CCRMA), USA. Größtenteils wurde es dann von ihm am Institut für Musik und Akustik, Zentrum für Kunst und Medientechnologie in Karlsruhe; Deutschland implementiert. Die Entwicklung setzt sich inzwischen fort an der University of Illinois, Urbana-Champaign, USA. 1996 gewann CM den ersten Preis in der Kategorie Computergestützte Komposition beim „1er Concours International de Logiciels Musicaux in Bourges“, Frankreich [vgl. Taube, a].

Über die Homepage von CM[6] sind viele weitere Links einschließlich einer Einführung[7] im CM erreichbar. Eine (etwas ältere) deutsche Einführung bietet Kunze [1994].[8] Die Referenz von CM stellt das Common Music Dictionary [Taube, b] dar. Über eine Mailingliste ist bei Problemen Hilfe durch die Entwickler möglich.[9] Das Programm einschließlich Dokumentation ist vom FTP-Server des CCRMA zu beziehen.[10] CM ist eine offene, freie Software, d.h. mit vollständigem Quellcode erhältlich, was für Erweiterungen bei der Entwicklung von Arno sehr wichtig war.

[2] Die graphische Oberfläche Capella [Taube and Kunze, 1995], existiert derzeit nur für *Macintosh Common Lisp* [Digitool Inc. Homepage].

[3] Neben MIDI werden insbesondere verschiedene Soundsynthesesprachen unterstützt: Csound, Common Lisp Music, Music Kit, C Mix, C Music, M4C, RT, Mix. Aber auch graphische Ausgabeformate können genutzt werden: VRML und Common Music Notation (und damit PostScript-Notensatz).

[4] Mit verschiedenen kommerziellen und auch freien Lispcompilern/-interpretern unter UNIX, MacOS, Windows u.a. wurden Common Music erfolgreich getestet. Eine genaue Auflistung ist bei Taube [a] zu finden.

[5] Eine Einführung in Pattern-Streams ist unter `http://ccrma-www.stanford.edu/CCRMA/Software/cm/items.html` zu finden, weitere Informationen finden sich bei Taube [b, Eintrag Item Streams].

[6] `http://ccrma-www.stanford.edu/CCRMA/Software/cm/cm.html`

[7] `http://ccrma-www.stanford.edu/CCRMA/Software/cm/intro/intro.html`

[8] Eine etwas überarbeitete Fassung ist unter `http://ccrma-www.stanford.edu/~tkunze/publ/dgm94/cm-dgm.html` zu finden.

[9] `cmdist-request@ccrma.stanford.edu`

[10] `ftp://ccrma-ftp.stanford.edu/pub/Lisp/cm/`

3.2.1. Wichtige Klassen in CM

CM ist mit den Mitteln von CLOS[11] objektorientiert programmiert: für die Partiturrepräsentation in CM wurde eine Hierarchie von Klassen definiert. Anhand dieser Hierarchie (siehe Abb. 3.1) stellt der folgende Abschnitt die Klassen vor, die im Kapitel 4 beim Benutzen von Arno eingesetzt werden.

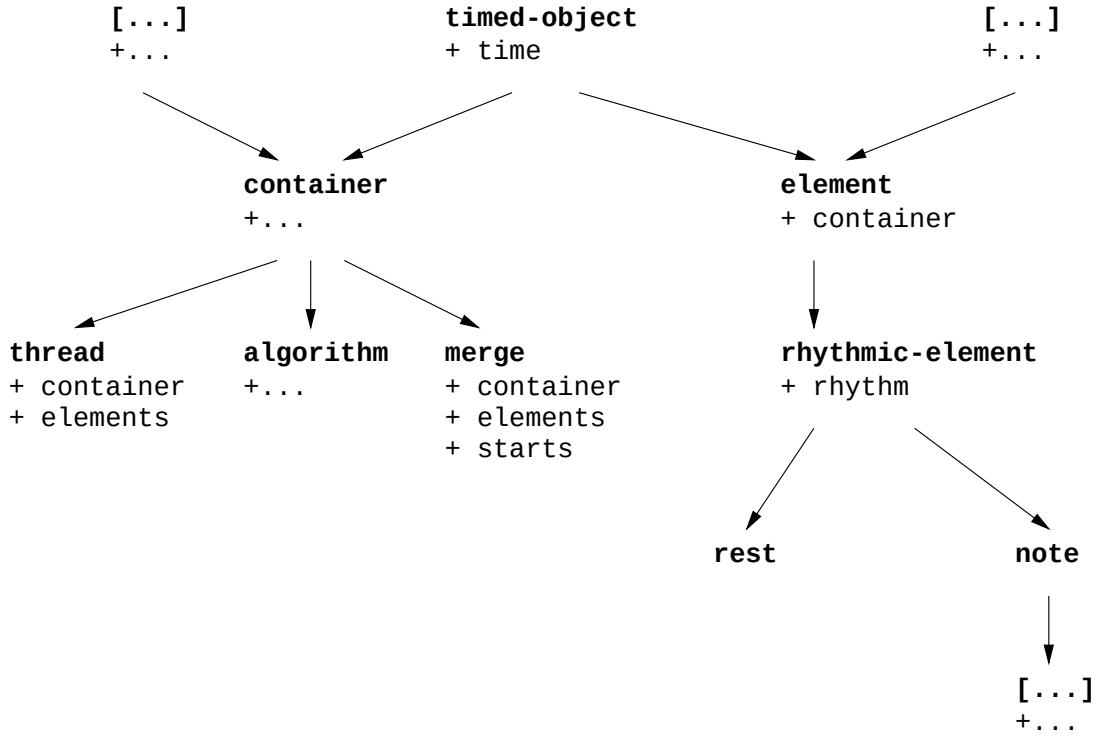

Abbildung 3.1.: Ein Ausschnitt aus der Klassenhierarchie in CM

Das ganz allgemeine `timed-object` verfügt bereits über den Slot `time` — allen davon abgeleiteten Klassen kann eine aktuelle Zeit zugewiesen werden.[12] Von `timed-object` leiten sich die beiden Klassen `element` und `container` her.

Ein Element stellt die Elternklassen für alle musikalischen Ereignisse dar: Noten, Pausen, auch Steuerinformationen (wie z.B. MIDI-Message). Elemente können in Containern enthalten sein, für diesen Zweck verfügen sie über den Slot `container`. Mit der Unterklasse `rhythmic-element` wird der Slot `rhythm` eingeführt, der die Zeitspanne bis zum Einsatz (`time`) des folgenden Elemente repräsentiert. Davon wiederum sind u.a. `rest` und `note` abgeleitet. Die Note-Klasse ist die Elternklasse verschiedener Klassen wie z.B. `midi-note`, `csound-note` und `clm-note`. Diese Klassen werden jeweils für verschiedene Ausgabeformate eingesetzt.

Ein Container dagegen kann verschiedenes enthalten, mit Hilfe von Containern kann eine Partitur strukturiert werden. Um genauer zu spezifizieren, was ein Container enthalten kann, und wie er diesen Inhalt verwaltet, werden von der Klasse `container` wiederum Unterklassen abgeleitet. In CM sind u.a. die Containerklassen `thread`, `merge` und `algorithm` definiert.

[11] Für eine Einführung in die Idee des objektorientierten Programmierens und Begriffe von CLOS siehe Anhang A.2.

[12] CM verwaltet die Zeitinformationen der Partitur, dieser Slot sollte als ein „read-only“-Slot verwendet werden.

Ein Thread enthält Objekte (Elemente und Container), gespeichert im Slot `elements`, die zeitlich sequenziell angeordnet werden, das heißt die Dauer eines Elements (`rhythm`) determiniert die Startzeit des folgenden Elements. Mit einem Thread läßt sich demnach eine Stimme darstellen. Auch Container innnerhalb Threads werden sequenziell angeordnet, ihre Position in der Zeit ist abhängig von der Dauer der in ihnen enthaltenen Elemente. Ein Thread kann in Containern enthalten sein, daher der Slot `container`.

Ein Merge enhält (im Slot `elements`) Container, denen jeweils ein eigener Startwert zugewiesen werden kann (Slot `starts`). Die in einem Merge enthaltenen Container können zeitlich parallel laufen, mit einem Threads läßt sich somit Mehrstimmigkeit repräsentieren. Auch ein Merge kann in Containern enthalten sein (Slot `container`).

Diese Beziehungen veranschaulicht Abb. 3.2.

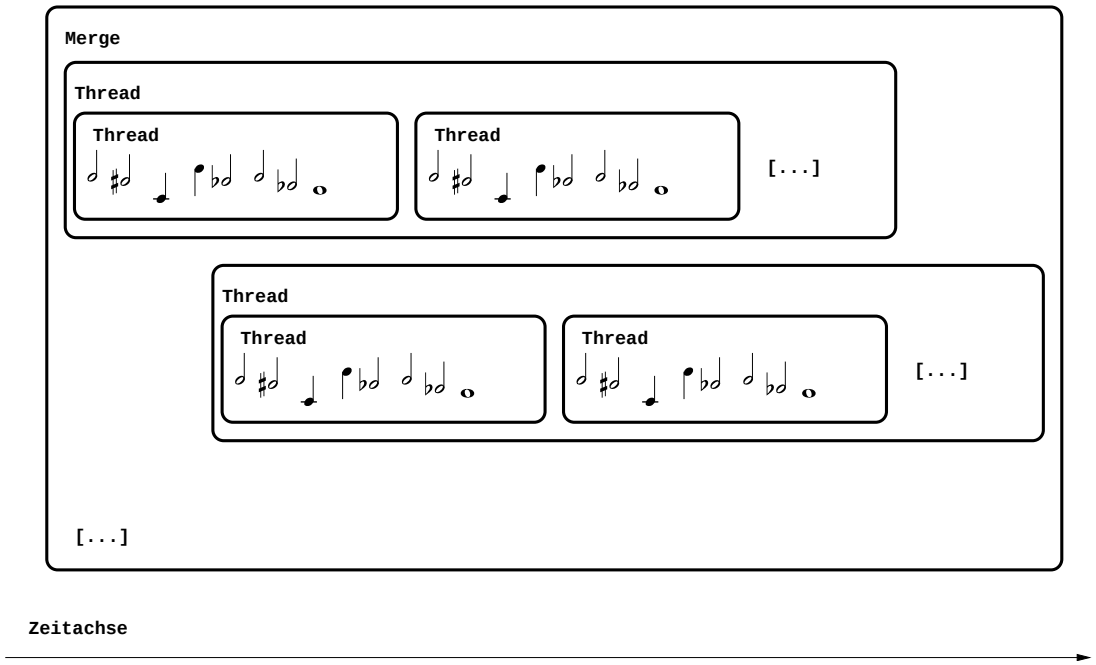

Abbildung 3.2.: Schematische Partiturrepräsentation in CM

Algorithms enthalten Code zur Erzeugung von Output, der bei jeder Abfrage neu evaluiert wird. Diese Containerklasse wurden für Arno nicht eingesetzt, da eine immer neu zu erfolgenden Evaluation zusammen mit nicht-deterministischen Ausdrücken jedesmal viel Rechenzeit erfordert.

3.2.2. Wichtige Befehle der CM-API

Die übliche Schnittstelle zur Benutzung von CM stellt die Shell Stella dar. Diese ist interaktiv leichter zu bedienen, als „puren“ Lisp-Code zu schreiben. Es sind z.B. viele Anweisungen in der Shell kompakter möglich, Objekte sind kompakter referenzierbar, es können Wildcards eingesetzt werden, fehlende Angaben eines Befehles werden interaktiv durch die Shell erfragt und es steht auch eine umfangreiche Hilfe zur Verfügung.

Um eine musikalische Struktur algorithmisch zu beschreiben wird in der Regel der Container Algorithm eingesetzt.

Soll aber CM nicht nur verwendet, sondern erweitert werden, so ist der Einsatz der Anwendungs-Schnittstelle (API) angezeigt, da mit dieser Kernfunktionen von CM unmittelbar zur Verfügung stehen. Im Rahmen dieser Arbeit war es nicht möglich, STELLA um die neudefinierte ARNO-Funktionalität zu erweitern, so daß auch der Benutzer von ARNO auf die Funktionen der CM-API angewiesen ist. Die verwendeten Befehle sind von Taube [b] dokumentiert, auch sind die Namen meist selbsterklärend. Deshalb wird hier nur ein Teil dieser Funktionen und dieser sehr kurz behandelt. In den Abbildungen 3.3 und 3.4 ist jede dieser Funktionen mit einem Beispiel kurz zusammengefaßt.

Das Makro `object` erzeugt eine neues Objekt eines gegebenen Typs, Slots können initialisiert werden. In der Darstellung der zurückgegebenen MIDI-Note sind die wichtigsten Parameter zu lesen: die Tonhöhe, die Dauer, der rhythmische Wert, die Amplitude (MIDI-Velocity) und der MIDI-Kanal. Die Makros `thread` und `merge` erzeugen einen entsprechenden Container. Es muß ein Name übergeben werden und weitere Objekte können als Inhalt initialisiert werden.

Ein Container kann mit seinem Namen mit Hilfe von `find-object` referenziert werde. Eine Kurzform davon ist #!. Der Inhalt eines Containers wird von `container-objects` zurückgegeben, `object-count` gibt die Anzahl der im Container enthaltenen Objekte zurück. Ein einzelnes Objekt innerhalb eines Containers ist mit `nth-object` zu erreichen.

```
? (object midi-note note 'c4 rhythm 2 duration 1 amplitude 120)
#<MIDI-NOTE | C4| 2| 1|120| 0|>

? (thread my-thread ()
    (object midi-note rhythm 1 note 60 channel 1)
    (object midi-note rhythm 1 note 62 channel 1)
#<THREAD: My-Thread>

? (find-object 'my-thread)
#<THREAD: My-Thread>
? #!my-thread
#<THREAD: My-Thread>

? (container-objects #!my-thread)
(#<MIDI-NOTE | 60| 1| 1| 64| 1|> #<MIDI-NOTE | 62| 1| 1| 64| 1|>)
? (nth-object 0 #!my-thread)
#<MIDI-NOTE | 60| 1| 1| 64| 1|>

? (object-count #!my-thread)
2
```

Abbildung 3.3.: Befehle der CM-API, durch Beispiele erläutert

Die Slots eines Elements können mit `set-object` gesetzt werden. Mit `careful-slot-value` können die Slots abgefragt werden.[13] Den Container, in dem sich ein Objekt befindet gibt `object-`

[13] Mit `careful-slot-value` können die Slots auch dann abgefragt werden, wenn sie nicht gesetzt sind. Dann

`container` zurück. Die Position des Objektes im Container ermittelt `object-position`.

```
? (set-object (object midi-note) 'rhythm 1)
#<MIDI-NOTE |---| 1| 1| 64| 0|>

? (careful-slot-value (object midi-note) 'note)
NIL

? (object-container (nth-object 0 #!my-thread))
#<THREAD: My-Thread>

? (object-position (nth-object 0 #!my-thread))
0
```

Abbildung 3.4.: weitere Befehle der CM-API

Zum Anzeigen und Bearbeiten eines mit Arno erzeugten Ergebnissen können die üblichen Befehle von Stella eingesetzt werden.

3.3. Screamer

Screamer [Siskind, 1991; Siskind and McAllester, 1993][14] ergänzt Common Lisp um eine Constraints-Programmiersprache zur Lösung numerischer und symbolischer Constraints, um Variablen ähnlich in Prolog, zurücknehmbare Nebenwirkungen (Side Effects)[15] u.a. Screamer ist vollständig in Lisp integriert und arbeitet mit anderen Lispextensionen wie Clos und Common Music zusammen.

Das von Screamer durchgeführte Back Tracking ist relativ effizient, da Screamer nicht-deterministische Ausdrücke vollständing in Common Lisp-Code transformiert, der von einem Compiler compiliert werden kann. Zum anderen erkennt Screamer nicht-deterministische Definitionen selbständig und transformiert nur diese entsprechend. Deterministische Ausdrücke bleiben unverändert.

Screamer ist hoch portabel[16] und die von Screamer zur Verfügung gestellte Funktionalität recht einfach einsetzbar. Screamer wurde von Jeffrey Mark Siskind und David Allen McAllester im Rahmen einer Lehrveranstaltung entwickelt und wird leider nicht weiter gepflegt. Die Dokumentation ist unvollständig und auch älter als die letzte Programmversion.

3.3.1. Nicht-deterministische Programmierung mit Screamer

Screamer erweitert Common Lisp in der Hauptsache um zwei Konstrukte. Das Makro `either` wählt nicht-deterministisch einen Ausdruck aus einer beliebigen Anzahl von übergebenen Aus-

wird NIL zurückgegeben. Die CLOS-Funktion `slot-value` dagegen erzeugt in einem solchen Fall eine Fehlermeldung.

[14] Die Quellen sind mit der Dokumentation zu beziehen beim Screamer-Resposity: http://www.cis.upenn.edu/~screamer-tools/home.html.

[15] Siehe S. 51.

[16] Eine Liste der getesteten Lispimplementierungen bietet `http://www.cis.upenn.edu/~screamer-tools/screamer-intro.html`.

drücken, evaluiert diesen und gibt das Ergebnis zurück. Mit der Funktion `fail` lassen sich einzelne nicht-deterministische Auswahlen untersagen. `either` wählt dann den nächsten Ausdruck aus der Auswahl. Nicht-deterministische Ausdrücke müssen in SCREAMER in einem Kontext stehen, der nicht-deterministische Ausdrücke zuläßt: `one-value` läßt solche Ausdrücke zu und gibt die erste gefundene Lösung zurück (Abb. 3.5) [vgl. Siskind and McAllester, 1993, S. 3].

```
? (one-value
    (let ((x (either 1 2 3 4)))
      (unless (> x 2)
        (fail))
      x))
3
```

Abbildung 3.5.: Das Zusammenspiel von `either` und `fail`.

Die LISP-Variabel `x` wurde im Beispiel nacheinander mit den Werten `1` und `2` gebunden, durch den Test wurde jedoch `fail` aufgerufen, was diese Bindungen unmöglich machte. Erst durch die Bindung von `x` mit 3 wurde `fail` nicht mehr aufgerufen, diese erste gefundene Lösung wurde zurückgegeben.

Nicht-deterministische Ausdrücke wie `either` können ineinander verschachtelt sein, `fail` erzwingt jeweils ein Back Tracking zum letzten Auswahlpunkt. Gibt es dort keine Alternativen mehr, wird ein BT zum davor liegenden Auswahlpunkt durchgeführt.

Innerhalb von `defun` sind ebenfalls nicht-deterministische Ausdrücke erlaubt, `defun` ist in Screamer umdefiniert. Um nicht-deterministische Ausdrücke in deterministische umzuformen, steht auch die Funktion `all-values` zur Verfügung, die eine Liste mit allen gefundenen Lösungen zurückgibt (Abb. 3.6).

```
? (defun demonstrate-bt ()
    (let ((x (either 1 (either 2 3 (either 4 5)))))
      (unless (> x 3)
        (screamer:fail))
      x))
DEMONSTRATE-BT
? (all-values (demonstrate-bt))
(4 5)
```

Abbildung 3.6.: Definition einer einfachen nicht-deterministischen Funktion

Mit `either` läßt sich recht leicht `an-integer-between` definieren, eine Funktion, die nicht-deterministisch eine Ganzzahl innerhalb der angegebenen Grenzen zurückgibt. Diese Funktion ist (wie z.B. auch `a-member-of`) bereits in SCREAMER vorhanden (Abb. 3.7). Funktionen wie `either`, `an-integer-between` oder `a-member-of` werden in SCREAMER als Generatoren bezeichnet.[17]

[17] Generatoren sind in SCREAMER durch den sie einleitenden unbestimmten Artikel im Namen kenntlich gemacht. Sie lassen sich als Ströme (*Stream*) auffassen, die eine Sequenz von Datenobjekten in einer festgelegten Reihenfolge ausgeben.

```
(defun an-integer-between (low high)
  (if (> low high) (fail))
  (either low (an-integer-between (1+ low) high)))
```

Abbildung 3.7.: Definition der SCREAMER-Funktion `an-integer-between`

Als ein etwas ausführlicheres Beispiel sei wieder das N-Queens-Problem angeführt (Abb. 3.8).[18] Natürlich muß dazu jede der Damen in einer anderen Spalte stehen, deshalb reicht zur Angabe der Lösung eine Liste mit den n möglichen Zeilenpositionen.

```
(defun attacks? (qi qj distance)
  (or (= qi qj) ; gleiche Zeile
      (= (abs (- qi qj)) distance))) ; gleiche Diagonale

(defun check-queens (queen queens &optional (distance 1))
  (unless (null queens)
    (when (attacks? queen (first queens) distance)
      (fail))
    (check-queens queen (rest queens) (1+ distance))))

(defun n-queens (n &optional queens)
  (if (= (length queens) n)
      queens
    (let ((queen (an-integer-between 1 n)))
      (check-queens queen queens)
      (n-queens n (cons queen queens)))))
```

Abbildung 3.8.: N-Queens-Problem gelöst mit SCREAMER

Das deterministische Prädikat `attacks?` testet, ob eine Damen-Zeilenposition (`qi`) von einer anderen Zeilenposition (`qj`), `distance` Spalten entfernt, angegriffen wird. `check-queens` untersucht, ob eine frisch gesetzte Position (`queen`) von keiner schon besetzten Position (`queens`) angegriffen wird, andernfalls wird `fail` aufgerufen. `n-queens` schließlich weist mit dem nichtdeterministischen Ausdruck `(an-integer-between 1 n)` einer Dame eine Position zu und ruft anschließend den Test auf. Ist diese Position nicht angegriffen, wird rekursiv die Lösung zusammengesetzt. Erfolgt ein `fail`, findet BT statt.

```
? (all-values (n-queens 4))
((3 1 4 2) (2 4 1 3))
```

3.4. Suche nach geeigneten Mitteln

Früh stand fest, daß ARNO auf COMMON MUSIC aufsetzen sollte. In CM sind sehr viele von ARNO benötigte Funktionalitäten wie die Partiturrepräsentation, die Ein- und Ausgabe in verschiedene

[18] [Vgl. Siskind and McAllester, 1993, S. 4]

Dateiformate usw. bereits realisiert. Zudem läßt sich CM durch seine offene API gut erweitern.

Für Arno sollte zusätzlich eine Umgebung gefunden werden, die Constraints-Programmierung bereits implementiert hat. Da für das Projekt keine zusätzlichen Gelder zur Verfügung standen und da Arno eventuell später weitergegeben werden soll, mußte eine freie Lösung gefunden werden.[19]

Es stellte sich nach ersten Versuchen mit CM und Prolog[20] schnell heraus, daß eine Implementierung in derselben Sprache wie CM äußerst wünschenswert sein würde, eine Anbindung über Pipes oder gar Dateiaustausch wäre zu umständlich geworden. Verglichen mit anderen Lisp-Umgebungen[21] ist Screamer (siehe Abschnitt 3.3) trotz seiner lückenhaften Dokumentation leichter zu bedienen, und auch eine relativ kleine Erweiterung.

Screamer wurde für Arno eingesetzt, da Screamer die einzige frei erhältliche Lisperweiterung ihrer Art darstellt. So konnte eine hohe Integration der verschiedenen Werkzeuge erreicht werden, die durch eine Kombination von Lisp mit Programmteilen in einer anderen Umgebung (wie etwa Prolog) zumindest mühsamer zu erreichen gewesen wäre.

3.5. Zusammenspiel der verschiedenen Umgebungen

3.5.1. Arno erweitert Common Music

Arno ergänzt die Kompositionsumgebung CM, d.h. die Funktionalität von Arno kann direkt aus CM heraus aufgerufen werden. Damit steht die reiche Palette an Common Music-Funktionen zur Generierung, Edierung, Ein- und Ausgabe verschiedener Dateiformate usw. zur Verfügung. Allerdings unterstützt Arno weder die CM-Shell Stella noch deren graphische Oberfläche Capella: mit Arno deklarierte Ausdrücke werden als Lispausdrücke evaluiert. Die so erzielten Resultate können aber mit Stella/Capella angezeigt, gespielt und bei Bedarf auch ediert werden. Arno erweitert CM mit Hilfe von Screamer um die Möglichkeit, ein gewünschtes musikalisches Resultat zu deklarieren, indem der Benutzer dieses lediglich genau beschreibt.

Da sowohl CM als auch Screamer hoch portabel sind und Arno nur Operatoren von Common Lisp, CM und Screamer einsetzt, sollte Arno vergleichbar portabel sein. Erfolgreich getestet wurde Arno mit *Allegro Common Lisp 4.3* [Franz Inc. Homepage] unter Linux und *Macintosh Common Lisp 4.0* [Digitool Inc. Homepage] unter MacOS.

[19] Mit Hilfe von Eikenberry, Jampel et al. [1996] (`http://www.cs.unh.edu/ccc/archive/constraints/systems/`) und Anfragen in einschlägigen Newsgroups (`comp.ai, comp.ai.shells`) ließen sich verschiedene Programmpakete finden, die eine freie Constraints-Programmierumgebung darstellen.

[20] Freie Prolog-Implementierungen sind z.B. SWI-Prolog (`ftp://swi.psy.uva.nl/pub/SWI-Prolog/`), B-Prolog (`http://www.cad.mse.kyutech.ac.jp/people/zhou/bprolog.html`) und GNU-Prolog (`http://pauillac.inria.fr/~diaz/gnu-prolog/`). Für ECLiPSe (eine Constraints-Entwicklungsumgebung abwärtskompatibel zu Prolog, `http://www.ecrc.de/eclipse/eclipse.html`) gibt es freie Lizensen.
Eine Einführung in Prolog bietet Bratko [1986].

[21] Babylon ist eine ExpertensystemImplementierung in Lisp [siehe Christaller et al., 1989].
Loom ist ein Wissensrepräsentationssystem in Lisp [siehe ISX, 1991, 1995].

3.5.2. Verschiedene Packages

Der Namensraum für Symbole (Namen von Funktionen, Variablen usw.) in Lisp ist in Packages organisiert, um Beschränkungen zu vermeiden und Überschneidungen zu verhinden.[22] CM und Screamer sind in je einer eigenen Package definiert. Da die Zahl Symbole in CM insbesondere auch durch die Shell Stella sehr groß ist, wurde eine Liste von zu exportierenden Symbolen nicht weiter gepflegt. Eine Funktion, die eine weitgehend vollständige Liste der CM-Funktionen zurückgibt, wurde auf Anfrage des Autors in der CM-Mailing-Liste dankeswerterweise von Heinrich Taube definiert und über die Mailing-Liste zur Verfügung gestellt.[23]

Um die Funktionalität von Screamer in einer anderen Package nutzen zu können, muß eine Screamer-Package definiert werden.[24] Für Arno wurde eine solche Screamer-Package definiert, die exportierten CM-Symbole wurde in diese Package importiert. Die wichtigsten Arno -Symbole wurden wiederum exportiert und nach CM importiert, um die Funktionalität von Arno auch direkt von Stella aus nutzen zu können.

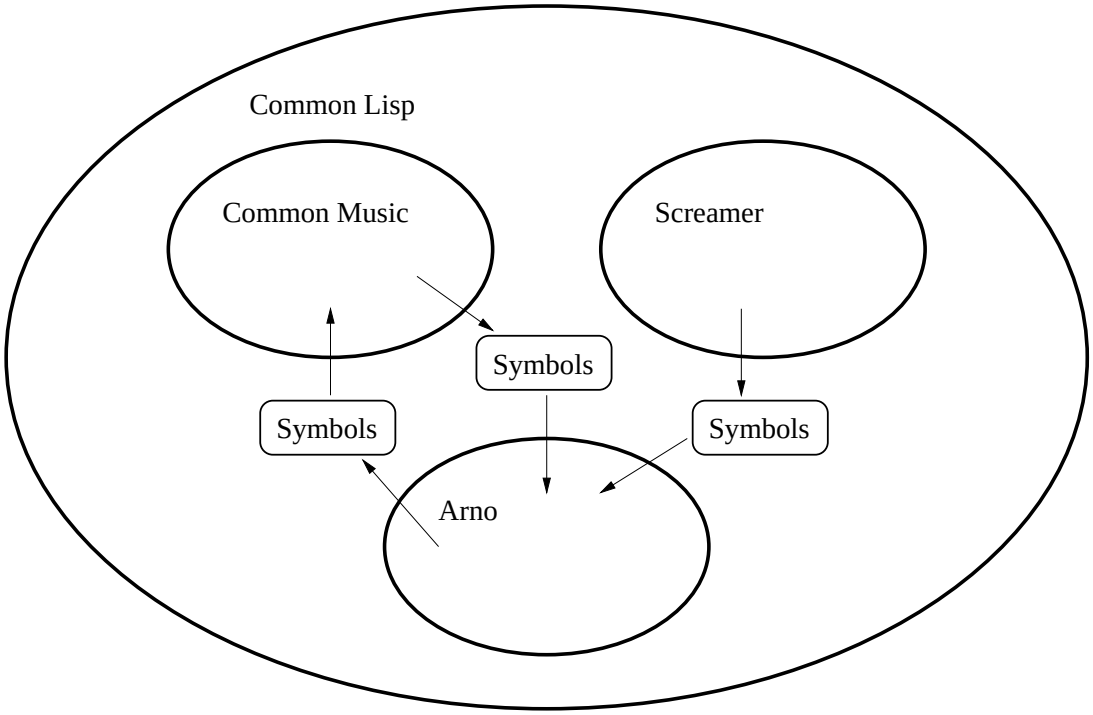

Abbildung 3.9.: Zusammenspiel der verschiedenen Packages

[22] Eine Einführung in Packages in Common Lisp bietet Graham [1994, S. 381 ff], die Referenz stellt Steele [1990, Kap. 11] dar.

[23] Für die Funktion `cm-exports` siehe C.7.

[24] Da in Screamer Primitive wie z.B. `defun` redefiniert wurden, müssen diese Redefinitionen auch in die neue Package übertragen werden (Shadowing Import), was durch die Definition einer Screamer-Package erreicht wird.

4. Die Anwendung von Arno

> Beim Entwerfen der Sätzchen geht der Schüler am besten systematisch vor, indem er [...] mit I, III beginnt und nach III anders fortsetzt. Also beispielsweise I, III, VI; nach VI kann dann einmal gleich I, das andere Mal IV und dann I folgen. Wenn alles mit I, III durchgearbeitet ist, beginne er mit I, IV, dann mit I, V und schließlich I, VI und setze jedesmal anders fort.
>
> Arnold Schönberg in seiner *Harmonielehre* [Schönberg, 1911, S. 46]

> Schoenberg [andere Schreibweise des Namens Schönberg im amerikalischen Exil, T.A.] liebte es, seine Arbeitsweise im Unterricht als „systematisches Vorgehen" zu bezeichnen; darunter verstand er die Methode, alle möglichen Lösungen der Reihe nach auszuprobieren. In der ersten Art des Kontrapunkts zum Beispiel brachte er jeden Ton des Cantus firmus systematisch mit allen möglichen Intervallen — mit der Prim, der Oktav, der reinen Quint, der Terz und der Sext — in Verbindung, Takt für Takt; dabei erklärte er die Vorzüge und Mängel jeder Kombination.
>
> Leonard Stein [Schoenberg, 1963, Vorwort, S. 7]

Das folgende Kapitel berichtet vom eigentlichen Gegenstand dieser Arbeit, von dem in ihrem Rahmen entwickelten Programm Arno. Nach einer Bemerkung zur Motivation der Entwicklung führt der Text an Hand eines Beispiels in die Handhabung ein. Es folgt eine Zusammenfassung der Möglichkeiten von Arno, ein etwas komplexeres Beispiel veranschaulicht diese weiter.

4.1. Motivation

Die Einleitung[1] führte bereits aus, daß das (auch) in der computergestützten Komposition verbreitete prozedurale Programmierparadigma dem Komponisten ernsthafte Einschränkungen auferlegt. Ein Komponist muß seine Arbeitsweise an dieses Paradigma anpassen: die Erzeugung und Modifikation der Daten für die einzelnen musikalischen Parameter erfolgt sequenziell. Ein Netzt von Bezügen zwischen verschiedenen Noten sind so schwer realisierbar. Man stelle sich vor, mit Zufallsgeneratoren, Prädikaten und Verzweigungen mehr als zwei Kontrapunktregeln implementieren zu wollen! Und wenn später auch nur eine Regeln geändert werden soll, ist fast ein Redesign des gesamten Programms nötig.

Arno erlaubt einem Komponisten, computergestützt zu komponieren, indem er Constraints — Eigenschaften, die die Musik erfüllen soll — deklariert. Die Prozedur der Lösungsfindung wird weitgehend vor dem Komponisten verborgen. Er kann sich darauf konzentrieren, *was* er musikalisch will, und das *Wie* dem Computer überlassen.

Arno soll mit möglichst freien kompositorischen Regelwerken umgehen können. Eine durch selbstdefinierte Regeln generierte Rhythmik ist ebenso realisierbar wie die Erzeugung von beliebig aufeinander bezogenen Syntheseparametern für ein Physical Modelling Instrument oder eine

[1] Siehe S. 6.

regelbasierte Harmonik mit Mikrotönen. Auch sollen diese verschiedenen musikalischen Aspekte — je nach Bedarf — voneinander abhängen können.

4.2. Ein erstes Beispiel

Ein kleines Beispiel (Abb. 4.1) demonstriert die Benutzung von Arno. Erzeugt werden acht Noten mit zufälligen Tonhöhen. Eine Constraint fordert, daß alle Töne einer Ganztonskala angehören. Es ist zunächst zu erkennen: Arno ist textbasiert. Der daraus resultierende etwas mühevollere Umgang erlaubt auf der anderen Seite eine große Flexibilität.

```
(defcontain zufalls-ganztoene
  :content-type (object midi-note rhythm 1)
  :number 8
  :content-setting
    (set-object (current-object)
                'note (a-shuffled-expr
                        (an-integer-between 60 72)))
  :fulfil '(in-ganztonleiter?))

(defmethod in-ganztonleiter? ((eine-note midi-note))
  ;; nur gerade Noten zulassen
  (evenp (slot-value eine-note 'note)))
```

Abbildung 4.1.: Zufällige Ganztöne

Das Beispiel enthält zwei Definitionen: eine Constraint (`in-ganztonleiter?`) und eine Funktion (`zufalls-ganztoene`), definiert mit dem Makro `defcontain`.

Eine Constraint wird in Arno als Prädikat definiert. Das Prädikat `in-ganztonleiter?` erwartet eine MIDI-Note und testet, ob deren Tonhöhe gerade ist und damit innerhalb der Ganztonleiter *c, d, e, fis...* liegt. Den eigentlichen Test stellt die Funktion `evenp` dar. Eine Constraint wird nacheinander auf jedes CM-Element in einem CM-Container angewendet.[2] Die Constraint erwartet deshalb genau ein Element als Argument, aus der der auszuwertende musikalische Parameter extrahiert wird: die CLOS-Funktion `slot-value` gibt den im jeweiligen Slot[3] enthaltenen Wert — hier im Slot `note` eine Zahl, die die Tonhöhe repräsentiert — zurück.

Mit dem Arno-Makro `defcontain` wird der Inhalt eines CM-Containers deklariert. Der Inhalt muß zunächst initialisiert werden: durch die den Argumenten `:content-type` und `:number` übergebenen Ausdrücke wird der Container mit acht MIDI-Noten gefüllt.[4] Dem Argument `:content-setting` wird ein Ausdruck übergeben, der diese Noten bearbeitet. Dieser Ausdruck enthält die Deklaration einer Domain für die Tonhöhe jeder der acht MIDI-Noten, was im nächsten Absatz weiter ausgeführt wird. Alle acht MIDI-Noten müssen der dem Argument `:fulfil`

[2] Für eine Einführung in die Mittel zur Partiturrepräsentation in Common Music (CM) siehe Abschnitt 3.2.1.

[3] Ein Slot ist eine Attributvariable eines CLOS-Objektes. Für eine Einführung in die Begriffe von CLOS, der objektorientierten Erweiterung von Lsip, siehe Anhang A.2.

[4] Mit der CM-Funktion `object` wird ein CM-Element erzeugt, siehe Abschnitt 3.2.2. Alle MIDI-Noten werden mit dem gleichen rhythmischen Wert 1 initialisiert, das entspricht einer Viertelnote bzw. bei Tempo 60 einer Sekunde.

übergebenen Constraint `in-ganztonleiter?` genügen: die Tonhöhen dieser Noten müssen innerhalb der Ganztonleiter $c, d, \ldots$ liegen. `defcontain` erzeugt eine Funktion, die bei Aufruf einen CM-Container als Argument erwartet. Zurück gibt sie eben diesen CM-Container, dessen Inhalt der mit `defcontain` erfolgten Deklaration entspricht.

Der dem Argument `:content-setting` übergebene Ausdruck enthält das Arno-Makro `current-object`. Dieses Makro evaluiert nacheinander zu allen Objekten im Container, es kann vergleichbar einer Variablen für alle Objekte im Container eingesetzt werden. Die CM-Funktion `set-object`[5] setzt durch den Einsatz von `current-object` die Tonhöhe (den Slot `note`) aller acht MIDI-Noten. Zu beachten ist, daß dies mittels der nicht-deterministischen Screamer-Funktion `an-integer-between`[6] geschieht, dadurch wird der Tonhöhe jeder MDI-Note eine Domain (60–72) zugewiesen. Der mit der Funktion `zufalls-ganztoene` gefüllte Container enthält MIDI-Noten, deren jeweilige Tonhöhe dieser Domain entstammt, die aber auch der Constraint `in-ganztonleiter?` genügen muß.

Die Arno-Funktion `a-shuffled-expr` bewirkt eine zufällige Anordnung der Werte in der Domain der Tonhöhe jeder einzelnen Note. Weil die Wahrscheinlichkeit, mit welchem Wert der Domain die Tonhöhe gebunden wird, von der Position dieses Wertes in der Domain abhängt, bewirkt die zufällige Anordnung der Domain, daß jede Tonhöhe mit einer zufälligen Tonhöhe gebunden wird (die der Constraint genügt).[7]

Das Argument `:fulfil` erwartet eine (quotierte) Liste: es können beliebige weitere Constraints definiert werden, die die Objekte im resultierenden Container erfüllen sollen.[8] Die zufälligen Ganztöne des Beispiels lassen sich prozedural einfacher erzeugen. Bei Arno aber ist ein Einbinden weiterer und ein Austauschen von Constraints möglich, ohne dazu jedesmal ein ganzes (prozedurales) Programm zu ändern.

In Abb. 4.2 wird (nach der Erzeugung des Containers `zufalls-ganztoene-container`) die mit `defcontain` definierte Funktion `zufalls-ganztoene` mit dem Container `zufalls-ganztoene-container` aufgerufen.[9] Die Funktion gibt diesen Container zurück, gefüllt mit acht zufälligen Ganztönen gemäß der Deklaration. Es ist jedoch nötig, den nicht-deterministischen Ausdruck (die Funktion enthält das nicht-deterministische `an-integer-between`) in einen deterministischen umzuwandeln. Deshalb ist beim Aufruf der Funktion `zufalls-ganztoene` ein vorangestelltes `one-value` erforderlich.[10]

Das Stella-Kommando `list` listet den Inhalt des Containers `zufalls-ganztoene-container` auf, zurückgegeben wird ein mögliches Ergebnis. Abb. 4.3 zeigt dieses Ergebnis übertragen in traditionelle Notenschrift.

[5] Funktion, um einen oder mehrere Slots eines Objektes zu setzen, siehe Abschnitt 3.2.2.

[6] `an-integer-between` gibt nicht-deterministisch eine Ganzzahl innerhalb des übergebenen Intervalls zurück, siehe Abschnitt 3.3.1.

[7] Diese Abhängigkeit von der Position eines Wertes in der Domain ist durch die in Arno verwendete Suchstrategie (Backtracking) verursacht, siehe Abschnitt 2.3.4.

[8] Die `fulfil` übergebene Liste muß quotiert sein, da diese durch einem Test, ob überhaupt Constraints mit `fulfil` übergeben wurden, evaluiert wird.

[9] Das CM-Macro `thread` erzeugt einen CM-Thread, das Lese-Makro `#!` aus CM gibt den Container mit dem übergebenen Namen zurück, siehe Abschnitt 3.3.

[10] Siehe Abschnitt 3.3.1.

```
Stella [Top-Level]: (thread zufalls-ganztoene-container ())
#<THREAD: Zufalls-Ganztoene-Container>
Stella [Top-Level]: (one-value (zufalls-ganztoene
                                #!zufalls-ganztoene-container))
#<THREAD: Zufalls-Ganztoene-Container>
Stella [Top-Level]: list Zufalls-Ganztoene-Container
Zufalls-Ganztoene-Container:
  1. #<MIDI-NOTE | 70| 1| 1| 64| 0|>
  2. #<MIDI-NOTE | 68| 1| 1| 64| 0|>
  3. #<MIDI-NOTE | 60| 1| 1| 64| 0|>
  4. #<MIDI-NOTE | 72| 1| 1| 64| 0|>
  5. #<MIDI-NOTE | 68| 1| 1| 64| 0|>
  6. #<MIDI-NOTE | 70| 1| 1| 64| 0|>
  7. #<MIDI-NOTE | 66| 1| 1| 64| 0|>
  8. #<MIDI-NOTE | 64| 1| 1| 64| 0|>
```

Abbildung 4.2.: Die Evaluation der Zufalls-Ganztöne

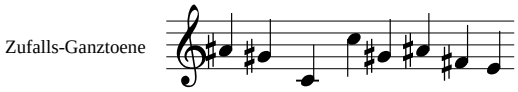

Abbildung 4.3.: Zufallsganztöne

4.3. Das Leistungsprofil von Arno

Das Grundanliegen der Entwicklung von Arno bestand darin, die Möglichkeiten der Constraints-Programmierung — spezialisiert auf die Generierung von Musik — dem Komponisten so zur Verfügung zu stellen, daß einerseits seine kompositorische Freiheit möglichst groß und andererseits die Nutzung möglichst einfach ist.

Der Nutzer von Arno trennt grundsätzlich die Deklaration in die Beschreibung der Form (im Beispiel: 8 MIDI-Noten) und die Definition der Constraints (im Beispiel: `in-ganztonleiter?`).

4.3.1. Freie Deklaration der Domain

Der Komponist deklariert (innerhalb von `:content-setting` im Makro `defcontain`) eine Domain grundsätzlich erlaubter Werte (jeweils für Tonhöhen, Dauern...), aus der durch die Angabe von Constraints eine Lösung gefiltert wird. Mit der Screamer-Funktion `a-member-of` können

jegliche Wertelisten und deren Kombinationen als Domain eines Parameters deklariert werden.[11]

4.3.2. Alle Parameter deklarierbar

Die Deklaration der Domain kann alle Parameter von CM-Elementen beschreiben, einschließlich derer für die Zeitorganisation (`rhythm`) und beliebiger Syntheseparameter.[12]

4.3.3. Constraints sind einstellige Prädikate

Constraints werden als einstellige Prädikate definiert. Eine Constraint kann jegliche Lispfunktion sein, die genau ein Argument — das aktuelle Element — erwartet und nach irgendeinem Test einen als Wahrheitswert interpretierbaren Wert zurück gibt.[13] Eine Constraint wird über ihren Namen in `defcontain` eingebunden. Constraints können je eine geforderte (`:fulfil`) oder zu vermeidende (`:avoid`) Eigenschaft testen. Constraints können unabhängig voneinander hinzugefügt, entfernt oder ausgetauscht werden.

4.3.4. Beliebige Relationen deklarierbar

Bei der Angabe einer Domain wie der Definition einer Constraint kann auf alle Parameter aller während des Suchvorganges bereits gesetzter Common Music-Elemente verwiesen werden. So kann mit Constraints gefordert werden, daß die Tonhöhe des gerade getesteten Tons bestimmten Stimmführungsregeln in Bezug zu(r) Vorgängernote(n) im selben Thread genügt, oder daß sie in die Harmonik (d.h. zu den Tonhöhen der bereits gebundenen simultanen Töne) paßt. Ebenso kann die Domain einer Tonhöhe von der Tonhöhe der Vorgängernote oder auch einem anderen Parameter derselben Note abhängen.

Da Arno keine zusätzliche interne Partiturrepräsentation benutzt, können bei der Referenzierung alle üblichen Befehle der CM-API verwendet werden. Diese ergänzt Arno um eine Reihe neuer Funktionen wie `previous-objects`, `simultan-objects` u.a., weitere können mit Hilfe der API ergänzt werden.

4.3.5. Verschachtelte Container bilden hierarchische Form

Die Beschreibung der Form kann hierarchisch gegliedert sein. So können z.B. mehrere kurze Threads mit je 3–5 Noten sukzessiv angeordnet werden in einem übergeordneten Thread, mehrere solcher übergeordnete Threads simultan wiederum eine Mehrstimmigkeit bilden (Threads in Merge) usf. Die Beschreibung der Form erfolgt mit dem Makro `defcontain`, dessen erstes Argument der Name der resultierenden Funktion ist. Über diesen Namen kann die resultierende Funktion in andere `defcontain`-Ausdrücke eingebunden werden (siehe Abschnitt 4.4.2).

[11] So sind mikrotonale Tonhöhen z.B. deklarierbar als Liste von Brüchen oder als je eine Liste erlaubter Zähler und Nenner.

[12] Aus der freien Deklarierbarkeit der Zeitstruktur ergeben sich Schwierigkeiten, die im Abschnitt 5.2.2 diskutiert werden.

[13] Lisp wertet die Konstante `NIL` (entspricht der leeren Liste) als unwahr, alle anderen Werte als wahr. Diese Eigenschaft ist für das Backjumping ausgenutzt worden (siehe S. 58).

4.3.6. Hüllkurven beschreiben Tongruppen

Da Musik nicht aus einer Anzahl von Einzeltönen besteht, sondern fast immer Gruppen von Tönen als Phrase wahrgenommen werden, bietet Arno die Verwendung von Hüllkurven zur Beschreibung von Parameterentwicklungen an.[14] Hüllkurven können zum einen für die Deklaration einer Domain eingesetzt werden. Zum anderen kann eine Constraintsdefinition Hüllkurven verwenden. In beiden Fällen kann den Objekten in einem Container je ein eigener Wert oder eine eigene Domain abhängig von ihrer Position im Container zugewiesen werden.

4.4. Ein komplexeres Beispiel

Ein weiteres Beispiel verdeutlicht einzelne Punkte der Möglichkeiten mit Arno. Obwohl Arno keinerlei stilistischen Beschränkungen unterliegt, werden im Beispiel verschiedene klassische Kontrapunktregeln definiert: da die Regeln selbst allgemein bekannt sind [siehe z.B. Jeppesen, 1930; Motte, 1981], kann der Fokus auf die Implementierung in Arno gerichtet sein.[15] Es werden zunächst die verwendeten Arno-Funktionen in einem Abschnitt zusammengefaßt vorgestellt (Abschnitt 4.4.1). Im folgenden Abschnitt wird die formale Struktur (Abschnitt 4.4.2) und anschließend eine Reihe von Constraints definiert (Abschnitt 4.4.3).

4.4.1. Verwendete Arno-Funktionen

Dieser Abschnitt stellt zusammengefaßt verschiedene, im Beispiel eingesetzte Arno-Funktionen vor. Einige Funktionen wurden schon beim ersten Beispiel (Abb. 4.1) eingesetzt, sind aber der besseren Übersichtlichkeit halber nochmal mit aufgeführt.[16] Zusätzlich werden im Beispiel auch Funktionen der CM-API verwendet, die im Abschnitt 3.2.2 eingeführt wurden.

defcontain funname &key **number content-type content-setting fulfil avoid let* further-args** => **funname** [Makro]

Das Makro `defcontain` definiert eine Funktion, die einen CM-Container als Argument erwartet, dessen eventuellen Inhalt verwirft und den Container füllt. Der gewünschte Inhalt des Containers kann nicht-deterministisch deklariert und durch Constraints näher bestimmt werden.

Die resultierende Funktion füllt den übergebenen Container mit einer Anzahl von `:number`[17] Objekten, die gemäß der Angabe in `:content-type` initialisiert werden. Die initialisierten Objekte können allen CM-Klassen mit einem `time`-Slot entstammen, der gewünschte Inhalt des Container kann damit aus CM-Elementen, aber auch aus CM-Containern bestehen.[18] In `:content-setting` kann jedem Slot initialisierter CM-Elemente nicht-deterministisch eine Domain von

[14] Dieses Konzept erlaubt die Beschreibung von Tendenzen statt Einzelnoten. Der Informationsgehalt der Tonhöhen einer Tonleiter beispielsweise läßt sich knapper mit Start- und Endwert einer gleichförmig linearen Hüllkurve beschreiben. Zudem ist dieses Konzept unabhängig von der tatsächlichen Anzahl der Töne.

[15] Das Beispiel bemüht sich nicht um eine Stilimitation, also etwa Josquin-Stil statt Palestrina-Stil. Demonstriert wird lediglich die Implementierung von typischen Regeln der klassischen Vokalpolyphonie.

[16] Die Funktionen werden z.T. nicht vollständig beschrieben, sondern nur soweit für das Verständnis des Beispiels nötig.

[17] Dem Argument `:number` kann derzeit kein Ausdruck übergeben werden. Soll die Anzahl evaluiert werden, muß `:number` mit dem Argument `:let*` eine lokale Variable übergeben werden (siehe `:let*` weiter unten).

[18] Dadurch ist eine hierarchisch gegliederte Form (Container in Containern) möglich.

Werten zugewiesen werden,[19] initialisierte CM-Container können ebenfalls bearbeitet werden. Die einzelnen Objekte sind mit der ARNO-Makro `current-object` erreichbar.[20] Jedes Objekt muß alle `:fulfil` übergebenen Constraints und darf keine `:avoid` übergebene Constraint erfüllen. Die Argumenten `:fulfil` und `:avoid` erwarten eine quotierte Liste.

Mit dem Argument `:let*` sind lokale Variablen innehalb der resultierenden Funktion definierbar, das Argument erwartet eine Liste im gleichen Format wie die LISP-Primitive `let*` und verhält sich entsprechend: schon deklarierte lokale Variablen können in folgenden Variablendeklarationen eingesetzt werden. Durch das Argument `:further-args` können in der resultierenden Funktion Argumente zusätzlich zu einem CM-Container definiert werden. `:further-args` erwartet diese Argumente in der gleichen Weise wie `defun`, d.h. in einer Lambda-Liste (*ordinary Lambda List*). Alle Lambda-Listen-Schlüsselworte (*lambda List Keywords*, z.B. `&optional`, `&rest`, `&key`) sind erlaubt.

Die resultierende nicht-deterministische Funktion muß innehalb eines Kontexts aufgerufen werden, der eine nicht-deterministische Funktion zuläßt. Solch ein Kontext ist eine Funktionsdefinition (mit `defun`, aber auch mit `defcontain`) oder eine Funktion, die einen nicht-deterministischen Ausdruck in einen deterministischen umwandelt (z.B. `one-value`).

current-object => timed-object [Makro]

Das Makro `current-object` evaluiert nacheinander zu allen Objekten im Container, es kann vergleichbar einer Variablen für alle Objekte im Container eingesetzt werden.[21]

let*-when ((var [init-form])*) form* => result [Makro]

Das Makro `let*-when` dient zur Definition lokaler Variablen und wird wie die LISP-Primitive `let*` eingesetzt. Allerdings evaluiert `let*-when` nur dann seinen Rumpf, wenn keine Initialisierungen einer deklarierten Variablen `NIL` ist, andernfalls wird `NIL` zurückgegeben (Abb. 4.4).[22]

previous-objects timed-object => timed-object-list [Funktion]

Die Funktion `previous-objects` ist eine Erweiterung der CM-API, um von einem gegebenen CM-Objekt aus zu vorangegangenen CM-Objekten referenzieren zu können. Die Funktion gibt eine Liste von Objekten zurück, die innerhalb eines Thread[23] sicher chronologisch vor dem als Argument übergebenen Objekt liegen, sich sozusagen in derselben Stimme befinden (Abb. 4.5). Die Reihenfolge der zurückgegebenen Objekte beginnt dabei beim unmittelbaren Vorgänger, es folgt dessen Vorgänger usf. Objekte im gleichen Merge (jeweils in Untercontainern) werden

[19] Dies kann z.B. mit der CM-Funktion `set-object` zusammen mit einem SCREAMER-Generator (wie `either` oder `a-member-of`) erfolgen.

[20] Siehe unten.

[21] Die aus defcontain resultierende Funktion iteriert in einer Schleife nacheinander durch alle Objektes im übergebenen Container: es werden nicht-deterministisch Werte zugewiesen und getestet (siehe Absatz 5.2.1). Mit `current-object` kann das jeweils „aktuelle“ Objekt innerhalb der Schleife erreicht werden.

[22] Die Primitive `let` und `let*` würden dagegen einen Fehler erzeugen, da der Rumpf immer evaliert wird. Das Makro `let*-when` ist das lediglich umbenannte `when-bind*` von Graham [1994, S. 145].

[23] Der Thread kann sich wiederum hierarchisch verschachtelt in anderen Containern befinden.

```
? (let*-when ((a 1)
              (b 2))
    (+ a b))
3
? (let*-when ((a 1)
              (b NIL))
    (+ a b))
NIL
```

Abbildung 4.4.: `let*-when` evaluiert Rumpf nur, wenn keine Variable `NIL` ist

nicht berücksichtigt, auch wenn sie sich zeitlich vor dem fraglichen Objekt befinden.[24] Wird `previous-objects` auf das erste Objekt angewendet, also ein Objekt, das keinen Vorgänger hat, gibt die Funktion `NIL` zurück.

previous-object timed-object => timed-object [Funktion]
previous-note timed-object => note [Funktion]

Die Funktionen `previous-object` und `previous-note` sind Spezialisierungen von `previous-objects`, die jeweils nur den unmittelbaren Vorgänger (`previous-object`) oder nur das unmittelbar vorangehende Objekt der CM-Klasse `note` (`previous-note`) zurückgeben.

simultan-objects timed-element => timed-element-list [Funktion]

Auch die Funktion `simultan-objects` ist eine Erweiterung der CM-API. Die Funktion gibt zu dem übergebenen Element gleichzeitige Elemente zurück (Abb. 4.6). Das sind Elemente, die vor dem fraglichen Element beginnen, deren Dauer aber in die Dauer des fraglichen Elements hineinragt, oder Elemente, die während der Dauer des fraglichen Elementes einsetzen. Gibt es keine gleichzeitigen Elemente, gibt die Funktion `NIL` zurück.

simultan-notes timed-object => note-list [Funktion]

Die Funktion `simultan-notes` ist eine Spezialisierung der Funktion `simultan-objects`. Zurückgegeben werden nur die simultanen Elemente der CM-Klasse `note`.

get-note note => note-slot-value [Funktion]
get-rhythm note => rhythm-slot-value [Funktion]

Mit den Funktionen `get-note` und `get-rhythm` können die entsprechenden Slots eines CM-Elementes gelesen werden.[25]

[24] Die ARNO-Funktion `all-previous-objects` gibt dagegen *alle* Objekte zurück, die zeitlich vor einem Objekt liegen.

[25] Durch den Einsatz der CM-Funktion `careful-slot-value` (siehe Abschnitt 3.2.2) in ihrer Definition geben sie `NIL` zurück, wenn der betreffende Slot nicht gesetzt ist.

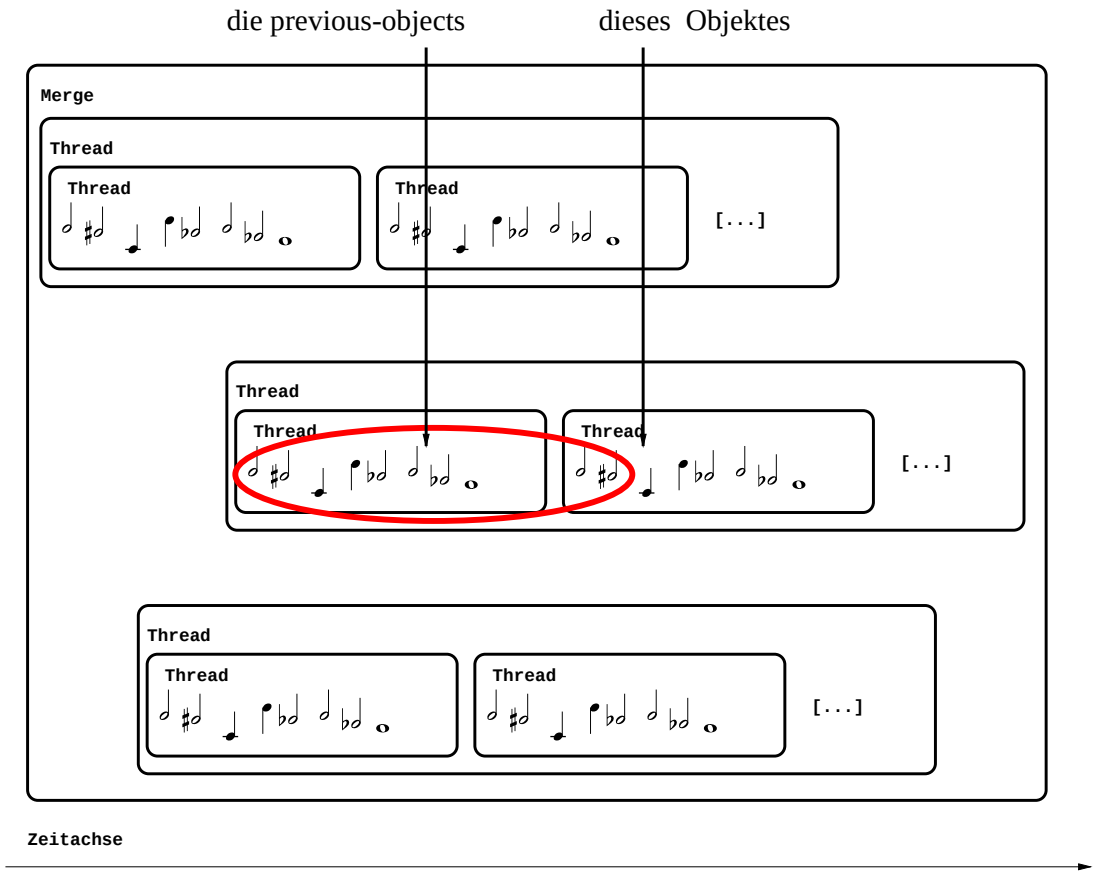

Abbildung 4.5.: Die `previous-objects` eines bestimmten Objektes

Hüllkurven in Arno

In Arno ist der Einsatz von Hüllkurven möglich.[26] Eine einfache Hüllkurve hat die Form

```
((zeit0 val0) <...> (zeit1 val1))
```[27]

mit `zeit0` = 0 (dies entspricht dem Anfang der Hüllkuve) und `zeit1` = 1 (das Hüllkurvenende). Zwischen diesen Begrenzungspunkten der Hüllkurve können (in aufsteigender Reihenfolge) beliebige und beliebig viele Zwischen-Zeitpunkte gesetzt werden. Jedem Zeitpunkt wird ein beliebiger Wert zugewiesen.[28]

[26] Vgl. Abschnitt 4.3.6.

[27] Die verwendetet Notation `<...>` steht für einen Ausdruck der ergänzt oder geändert werden kann.

[28] Es sind alle Lisp-Ausdrücke, einschließlich nicht-deterministischer Ausdrücke, erlaubt. Für den Einsatz von Ausdrücken ist jedoch eine ausführlichere Syntax nötig in der Form:

```
(list (list zeit0 <ausdruck0>)
      <...>
      (list zeit1 <ausdruck1>))
```

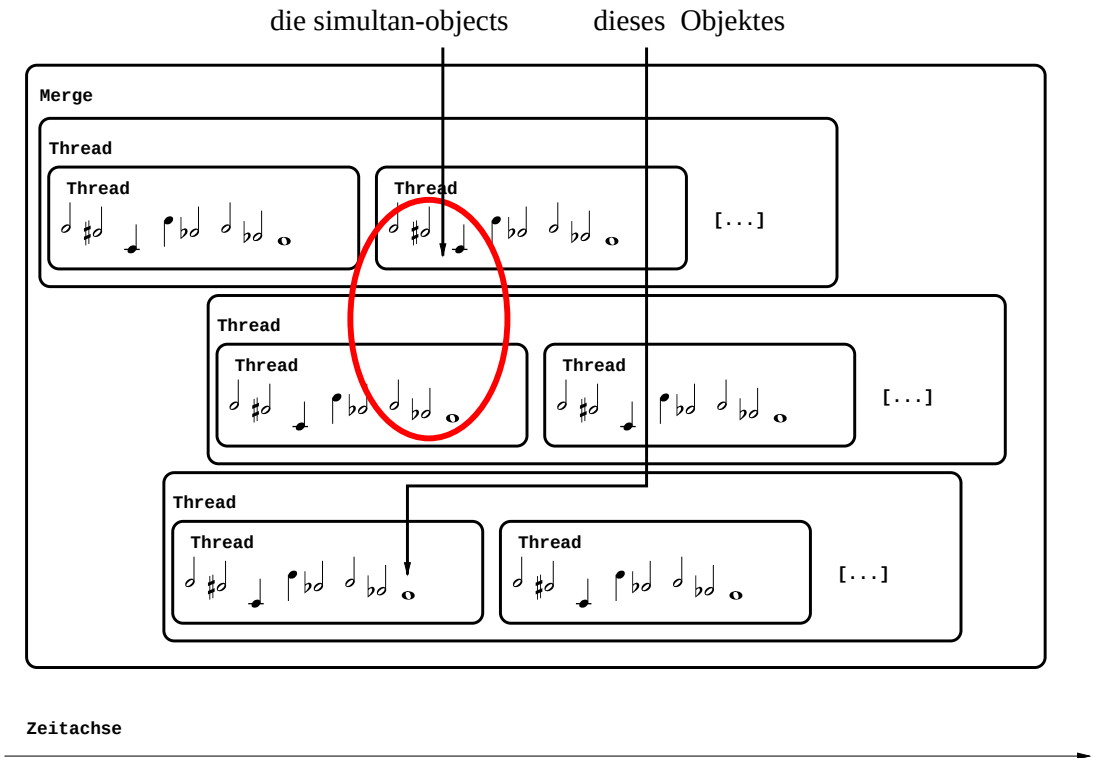

Abbildung 4.6.: Die `simultan-objects` eines bestimmten Elementes

env-value i length env => curr-env-value [Funktion]

Mit die Funktion `env-value` kann der Wert einer Hüllkurve an einer bestimmten Stelle ermittelt werden. Um für jedes einzelne CM-Element in einem CM-Container bequem einen Wert bestimmen zu können, ist die Idee einer Hüllkurve mit diskreten Werten zugrunde gelegt. `env-value` erwartet den Index des gewünschten Wertes (zählend ab 0), die Anzahl aller Hüllkurvenwerte[29] und die Hüllkurve. Die Funktion gibt den Hüllkurvenwert an der Index-Position zurück.

```
? (env-value 0 3 '((0 1) (.7 2) (1 4)))
1
? (env-value 1 3 '((0 1) (.7 2) (1 4)))
1.4761906
```

upper-env double-env => upper-env [Funktion]
lower-env double-env => lower-env [Funktion]

Mit der Deklaration zweier, parallel laufender Hüllkurven ist ein über die Zeit veränderlicher Wertebereich definierbar. Eine solche doppelte Hüllkurve kann mit der Syntax

```
((zeit0 val0a val0b) <...> (zeit1 val1a val1b))
```

[29] Um mehrere Hüllkurven nahlos aneinander fügen zu können, werden die Werte bis vor den letzten Zeitwert (1) gezählt. Der Wert der Hüllkurve beim Zeitwert 1 wird erst mit Index=Anzahl der Werte erreicht, dieser Wert kann gleichzeitig Zeitwert 0 einer anschließenden Hüllkurve sein.

ausgedrückt werden. Die beiden einzelnen Hüllkurven lassen sich mit den Funktionen `upper-env` und `lower-env` extrahieren, die jeweils die obere bzw. untere Hüllkurve zurückgeben.

```
? (upper-env '((0 1 2) (.3 0 4) (1 1 1)))
((0 2) (0.3 4) (1 1))
? (lower-env '((0 1 2) (.3 0 4) (1 1 1)))
((0 1) (0.3 0) (1 1))
```

4.4.2. Die Deklaration einer hierachischen Form

In Abb. 4.7 wird mit `defcontain` die Funktion `stimme` definiert. Die CM-Funktion `between` gibt einen zufälligen Wert in dem übergebenen Intervall zurück: es sind 11–17 MIDI-Noten gewünscht. Der `rythm`-Slot der MIDI-Noten wird mit 0 initialisiert.[30]

```
(defcontain stimme
  ;; resultierende Funktion erwartet Thread als Argument
  :further-args (&key min-ton max-ton channel)
  :let* ((number (between 11 17)))
  :content-type (object midi-note rhythm 0)
  :number number
  :content-setting
  (set-object (current-object)
    'rhythm (a-shuffled-expr (a-member-of '(16 8 6 4 3 2 1)))
    ;; unary constraint: c-major bei gegebenem Stimmumfang
    'note (note (mode-degree
                 (a-shuffled-expr
                   (an-integer-between min-ton max-ton))
                 c-major))
    'channel channel))
   :avoid '(<...>) )

(defvar c-major (transpose (mode major 2 2 1 2 2 2 1) 'c))
```

Abbildung 4.7.: Die Definition der Funktion `stimme`

In `:content-setting` wird, wie im ersten ARNO-Beispiel (Abb. 4.1), die CM-Funktion `set-object` in Kombination mit dem ARNO-Makro `current-object` und SCREAMER-Generatoren eingesetzt, alle im Container enthaltenen MIDI-Noten werden dadurch ediert. Es werden neben dem Slot `note` auch die Slots `rhythm` und `channel` gesetzt: in gleicher Weise ist jeder Slot des `current-object` deklarierbar (vgl. Abschnitt 4.3.2).

Weiterhin ist zu erkennen, daß die Tonhöhen-Domain mit `mode-degree`[31] auf Tonhöhen der

[30] Der Rhythmus aller Elemente muß mit einem numerischen Wert initialisiert sein. Dessen Größe ist nicht von Bedeutung, wenn er (wie hier) später überschrieben wird.

[31] Die Funktion `mode-degree` quantisiert die als erstes Argument übergebenen Tonhöhen in die als zweites Argument übergebene Tonart, hier C-Dur. Die Funktionen `mode-degree`, `transpose` und `mode` sind in CM implementiert [siehe Taube, b].

C-Dur-Skala reduziert wird. Die Tonhöhen-Domain wird dadurch so reduziert, daß sie in Bezug auf eine (nicht definierte) Constraint `in-c-dur?` knoten-konsistent ist.[32]

Das Argument `:avoid` erwartet eine quotierte Liste von Constraints, die nicht erfüllt sein dürfen. Diese werden im folgenden Abschnitt 4.4.3 definiert und sind deshalb hier nicht namentlich alle aufgezählt.

Eine hierarchische Gliederung der Form erfolgt durch die Definition von `zweistimmig` (Abb. 4.8). Die initialisierten Objekte sind zwei CM-Threads.[33] In `:content-setting` wird die Funktion `stimme` jeweils mit einem dieser Threads (dem `current-object`) aufgerufen. `zweistimmig` wiederum wird später mit einem CM-Merge aufgerufen: dadurch laufen die beiden jeweils mit `stimme` gefüllten Threads in der Zeit parallel. Um jeder Stimme einen eigenen Stimmumfang und MIDI-Kanal übergeben zu können, wird die Position des jeweiligen Threads im Merge ausgewertet (vgl. Abschnitt 4.3.5).

```
(defcontain zweistimmig
  ;; erwartet Merge als Argument
  :content-type (make-object 'thread)
  :number 2
  :content-setting
  (let ((curr-obj (current-object)))
    (case (object-position curr-obj)
      (0 (stimme curr-obj
                 :min-ton 48 :max-ton 65 :channel 0))
      (1 (stimme curr-obj
                 :min-ton 50 :max-ton 67 :channel 1))) ))
```

Abbildung 4.8.: Eine hierarchische Formdefinition durch Aufruf von `stimme` in `zweistimmig`

Mit diesen beiden Funktionen wurde das gewünschte formale Grundgerüst definiert: zwei parallele Stimmen, jede enthält eine gewisse Anzahl von MIDI-Noten. Außerdem wurde bereits eine einstellige Contraints formuliert: alle MIDI-Noten sollen der C-Dur-Skala angehören. Schließlich wurde für die Domain der Tonhöhen und die der rhythmischen Werte jeweils eine (schon im Beispiel Abb. 4.1 eingesetzte) Heuristik hinzugefügt: die Reihenfolge der Werte soll zufällig sein (`a-shuffled-expr`).[34]

4.4.3. Verschiedene Constraintsdefinitionen

Im folgenden Abschnitt werden eine Reihe von Constraints definiert. Alle lehnen sich an traditionelle Kontrapunktregeln an, demonstrieren aber vor allem verschiedene Möglichkeiten für die Contraints-Programmierung in Arno.

Eine *einstellige Constraint*, d.h. eine Constraints, die sich nur auf das `current-object` bezieht, wurde bereits im Beispiel Abb. 4.1 gezeigt: `in-ganztonleiter?` schränkte nur das jeweilige

[32] Siehe S. 13.
[33] Die CM-Funktion `make-object` erwartet einen Objektyp (und eventuell Slot-Initialisierungen) [siehe Taube, b].
[34] Siehe Abschnitt 2.3.4.

`current-object` weiter ein. Eine weitere Möglichkeit, einstellige Constraints zu definieren, wurde innerhalb der Definition von `stimme` in Abb. 4.7 demonstriert: mit `mode-degree` wurde die Tonhöhen-Domain reduziert.

Eine *binäre Constraint* kann das das `current-object` und das diesem vorangehende Objekt in derselben Stimme aufeinander beziehen. So testet die Constraint `nicht-erlaubte-Intervalle?`, ob das Intervall zum vorhergehenden Ton in derselben Stimme klassischen Stimmführungsregeln genügt (Abb. 4.9).

```
(defmethod nicht-erlaubte-Intervalle? ((note midi-note))
  (let*-when ((prev (previous-note note))
              (intervall (- (degree (get-note note))
                            (degree (get-note prev)) )))
    ;; Intervall muß größer als Prim und kleiner als Quart sein
    ;; oder ist Quint bzw. Oktav
    ;; oder (nur aufwärts) ist kleine Sext
    (unless (or (< 0 (abs intervall) 5)
                (member (abs intervall) '(7 12))
                (= intervall 8))
      t)))
```

Abbildung 4.9.: Constraints mit Bezug zur vorigen Note im Container

Diese Constraintsdefinition folgt demselben Programmiercliché[35] wie alle folgenden Definitionen auch. Dieses Cliché setzt `let*-when` ein. Mit `let*-when` werden lokale Variablen deklariert und mit einem Ausdruck initialisiert. In Abb. 4.9 z.B. wird die Variable `prev` mit `(previous-note note)` initialisiert. Evaluiert der Initialisierungsausdruck mindestens einer Variablen in `let*-when` zu `NIL`, ist z.B. für die erste Note eines Threads kein Vorgänger vorhanden, ist auch der Wert von `let*-when` `NIL` und die Constraint gibt `NIL` zurück.

Die Constraints werden alle in die Funktion `stimme` (Abb. 4.7) als *nicht* zu erfüllende Constraints eingebunden (`avoid:`). Eine nicht zu erfüllende Constraint, die `NIL` zurückgibt, ist damit erfüllt.

Evaluiert keine der Initialisierungen in `let*-when` zu `NIL`, wird der mit `unless` formulierte Test durchgeführt. Wird der Test *nicht* erfüllt, gibt die Constraint `T` zurück und ist damit nicht erfüllt. D.h. enthält die aktuelle Teillösung nicht erlaubte Intervalle (`nicht-erlaubte-Intervalle?` gibt `T` zurück) muß eine andere Lösung gefunden werden.[36]

In der Definition in Abb. 4.9 wird die CM-Funktion `degree` eingesetzt. In CM können Tonhöhen durch einen Notennamen (als Symbol), durch eine Frequenz (als Fließkommazahl) oder durch einen MIDI-Tonhöhenwert (als Ganzzahl) angegeben werden. `degree` erwartet eine Tonhöhenbezeichnung in einer der drei möglichen Formen und gibt den zugehörigen MIDI-Wert zurück.

[35] Mit Programmiercliché wird eine Art Schablone für die Programmierung einer Prozedur oder Teilprozedur bezeichnet, die allgemein verwendbar ist.

[36] Alle im folgenden definierte Constraints dürfen nicht erfüllt sein, da sie dem `defcontain`-Argument `:avoid` zugewiesen werden. Die etwas umständliche Formulierung als negative Constraint wurde gewählt, um die gleiche Constraint auch für eine Suche mit Backjumping einsetzen zu können (vgl. S. 58 und Anhang B). Für Backjumping sind `:avoid`-Constraints erforderlich, die bei Nichterfüllung das Objekt zurückgeben, mit dem die Constraint versagte. Diese Objekte können dann Rücksprungziele sein.

Eine *mehrstellige Constraint* kann das `current-object` auf die Vorgängernote und deren Vorgängernote beziehen (Abb. 4.10). Die Constraint `no-gegenbew-nach-gr-sprung?` wertet die beiden Intervalle zwischen den letzten drei Noten aus.

```
(defmethod no-gegenbew-nach-gr-sprung? ((note midi-note))
  (let*-when ((prev (previous-note note))
              (prev-prev (previous-note prev))
              (int-1 (- (degree (get-note prev))
                        (degree (get-note prev-prev)) ))
              (int-2 (- (degree (get-note note))
                        (degree (get-note prev)) )))
    ;; wenn erstes Intervall groesser als Quint,
    ;; muss naechstes Intervall entgegen sein:
    ;; max. kl. Terz nach aufwaerts Sprung,
    ;; max. gl. Intervall wieder hoch nach Abwaertssprung
    (unless
      (cond ((> int-1 7) (<= -3 int-2 0))
            ((< int-1 -7) (<= 0 int-2 int-1))
            (T T))
      t)))
```

Abbildung 4.10.: Constraints mit Bezug zu den beiden vorigen Noten im Container

Ebensogut kann eine Constraint noch mehr Noten der gleichen Stimme einbeziehen und diese Noten müssen auch keine unmittelbaren Vorgänger sein. Die Constraint `dupl-peaks?` (Abb. 4.11) wertet die Wendepunkte in einer Stimme (numerisch die lokalen Maxima und Minima) aus. Diese Wendepunkte ermittelt die in Abb. 4.12 definierten Funktion `letzter-richtungswechsel`. Die Constraint `dupl-peaks?` stellt sicher, daß zwei unmittelbar aufeinander folgende lokale Maxima bzw. zwei solche Minima nicht denselben Wert haben, eine klassische Kontrapunktregel, wenn auch traditionell nicht derart streng gefaßt.

```
(defmethod dupl-peaks? ((note midi-note))
  (let*-when ((prev-turn (letzter-richtungswechsel note))
              (prev-prev-turn (letzter-richtungswechsel prev-turn))
              (prev-prev-prev-turn
                (letzter-richtungswechsel prev-prev-turn)))
    ;; die Tonhoehen zweier Wendepunkte in die gl. Richtung
    ;; sollen sich nicht gleichen
    ;; (deshalb einer ausgelassen)
    (when (= (degree (get-note prev-turn))
             (degree (get-note prev-prev-prev-turn)))
      prev-turn)))
```

Abbildung 4.11.: Contraints mit Bezug auf mehrere Vorgänger-Noten.

Die etwas kompliziertere Constraint `nicht-erlaubter-zsklang?` (Abb. 4.13) demonstriert einen Bezug auf simultane Elemente des `current-object`. Alle gleichzeitigen Töne sollen eine vollkommene oder eine unvollkommene Konsonanz zur Bezugsnote bilden. In der gezeigten Formulierung

```
(defmethod letzter-richtungswechsel ((note midi-note))
  ;; wo letzter Vorzeichenwechsel der Folgeintervalle
  ;; in Stimme vor note?
  (let*-when ((prev (previous-note note))
              (prev-prev (previous-note prev))
              (int-1 (- (degree (get-note prev))
                        (degree (get-note prev-prev)) ))
              (int-2 (- (degree (get-note note))
                        (degree (get-note prev)) )))
    (if (or (and (< int-1 0) (> int-2 0))
            (and (> int-1 0) (< int-2 0)))
        prev
      (letzter-richtungswechsel prev))))
```

Abbildung 4.12.: Die Funktion `letzter-richtungswechsel`

ist die Constraint nur für zweistimmigen Kontrapunkt einsetzbar, da die Quarte nicht gesondert behandelt wird. Es sind mit dieser Constraint auch keinerlei Durchgangsdissonanzen oder Vorhalte gestattet. Hier sollte aber vor allem gezeigt werden, wie eine Constraint formuliert werden kann, die gleichzeitige Elemente aufeinander bezieht. Verfeinerungen hätten das Beispiel weiter verkompliziert.

Mit den Beispielen Abb. 4.10–4.13 wurde gezeigt, wie in einer Constraint Relationen zwischen beliebigen Elementen definiert werden können (vgl. Abschnitt 4.3.4). Die Definition der Constraint `no-smooth-rhythm?` (Abb. 4.14) zeigt, daß sich Constraints auf alle Parameter dieser Elemente beziehen können (vgl. Abschnitt 4.3.2). Diese Constraints bezieht sich auf die `rhythm`-Slots des `current-object` und seine Vorgängers. `no-smooth-rhythm?` erzwingt einen allmählichen Übergang von langen zu kurzen rhythmischen Werten (und umgekehrt), die rhythmischen Werte aufeinanderfolgender Noten dürfen höchsten das Dreifache und müssen mindestens ein Drittel des vorangehenden Wertes betragen.

Die Constraint `not-in-rhythm-env?` (Abb. 4.15) schließlich setzt eine doppelte Hüllkurve ein, um einen erlaubten Wertebereich für rhythmische Werte zu definieren, der von der Position des jeweiligen Objektes und der Anzahl der Objekte im Container abhängig ist (vgl. Abschnitt 4.3.6).[37]

Mit der im Anhang B mitgeteilten etwas ausführlicheren Fassung eines kontrapunktischen Regelwerks lassen sich Ergebnisse wie z.B. in Abb. 4.16 erzielen.

[37] Diese Constraint läßt sich auch in der Deklaration der `rhythm`-Domain in `defcontain` einbinden — ähnlich der Reduzierung der Tonhöhendomain auf Tonhöhen aus C-Dur — was eine erhöhte Effizienz zur Folge hat. Der besseren Übersichtlichkeit wegen ist `not-in-rhythm-env?` aber gesondert definiert.

```
(defmethod nicht-erlaubter-zsklang? ((note midi-note))
  (let*-when ((note-degree (degree (get-note note)))
              (simultans (remove-if-not
                          #'(lambda (simultan)
                            ; ist schon Notenwert gesetzt?
                             (careful-slot-value
                               simultan 'note))
                         (simultan-notes note)))
              (simultan-dissonances
                (mapcar #'(lambda (simultan)
                            (let ((interval
                                    (- (degree (get-note simultan))
                                        note-degree)))
                              ;; wenn keine Konsonanz
                              ;; (gegebene Intervalle),
                              ;; gib simultan zurueck, sonst NIL
                              (unless (member (abs interval)
                                        '(3 4 7 8 9 12 15 16))
                                        simultan)))
                        simultans)))
    ;; wenn irgendeiner nicht NIL, gib den ersten zurueck
    (when (or simultan-dissonances)
      (first (remove-if #'null simultan-dissonances)))))
```

Abbildung 4.13.: Constraints mit Bezug auf simultane Elemente

```
(defmethod no-smooth-rhythm? ((note midi-note))
  ;; naechste note mindestens 1/3 und
  ;; hoechsten dreimal so lang wie Vorgaenger
  (let*-when ((prev (previous-note note)))
    (unless (<= 1/3
                (/ (get-rhythm prev)
                   (get-rhythm note))
                3)
      prev)))
```

Abbildung 4.14.: Constraint bezogen auf den Rhythmus

```
(defmethod not-in-rhythm-env? ((note midi-note))
  ;; um die globale Dauernentw. kontrollieren zu koennen
  (let* ((rhyhtm-env '((0 8 16)
                       (.2 2 8)
                       (.5 1 6)
                       (.7 1 4)
                       (.8 1 6)
                       (.9 2 8)
                       (1 8 16)))
         (max-env (upper-env rhyhtm-env))
         (min-env (lower-env rhyhtm-env))
         (note-anzahl (object-count (cm::the-container note)))
         (note-nr (object-position note))
         (max-env-wert (env-value note-nr note-anzahl max-env))
         (min-env-wert (env-value note-nr note-anzahl min-env)))
    (not (<= min-env-wert (get-rhythm note) max-env-wert))))
```

Abbildung 4.15.: Constraint unter Einsatz einer Hüllkurve

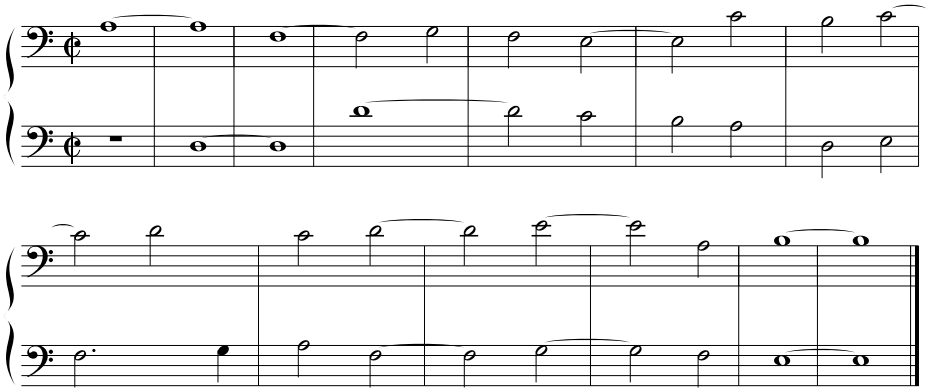

Abbildung 4.16.: Ein kleines zweistimmiges Kontrapunktbeispiel

5. Die technische Realisierung von Arno

Das folgende Kapitel skizziert die technische Realisierung von Arno. Nach einer kurzen Einleitung wird im Abschnitt 5.2.1 die Implementierung des Makros `defcontain` erläutert. Dazu zeigt der Text zunächst an Hand eines verwandten Arno-Makros (`defcsp`), wie in Lisp eine Funktion durch ein Makro definiert werden kann. Darauf wird das Konzept lokaler Nebenwirkungen (Side-Effects) in Screamer eingeführt. Es wird demonstriert, wie mit Hilfe lokaler Nebenwirkungen in einem nicht-deterministischen Ausdruck eine Schleife verwendet werden kann, und wie dies die Definition von `defcontain` einsetzt. Kurz weist der Text auf eine Einsatzweise von `let*` in `defcontain` hin, durch die die resultierende Funktion nicht unnötig groß wird.

Im folgenden Abschnitt 5.2.2 werden Schwierigkeiten genannt, die sich aus der nicht vordefinierten Zeitstruktur der Objekte im resultierenden Container ergeben.

Der Absatz 5.3 schließlich skizziert die Implementierung von nötigen Erweiterungen der CM-API.

5.1. Einleitung

Arno ist durch Constraints-Programmierung realisiert, das Programm führt eine Suche mit Backtracking durch. Diese Technik verbindet eine große Ausdrucksstärke und Flexibilität in der Anwendung mit relativ einfacher Implementierung. Die geringe Ausführungsgeschwindigkeit einer blinden Suche wurde in Kauf genommen. Überlegungen zur Effizienzsteigerung werden im Abschnitt 6.2 diskutiert.

5.2. Das Makro `defcontain`

5.2.1. Die Implementierung

Dem eigentlichen Kern von Arno, dem Makro `defcontain`, liegt das in Abb. 5.1 mitgeteilte Programmiercliché zugrunde. Definiert wird eine Funktion, die für die Bindung der übergebenen CSP-Variable sicherstellt, daß alle in einer Liste übergebenen Constraints (jeweils als einstelliges Prädikat definiert) eingehalten sind. Dann wird der Wert der Variablen zurückgegeben.[1]

Die Arno-Funktion `all-true?` erwartet zusammen mit dem zu testenden ersten Argument eine Liste von einstelligen Prädikaten. Die Prädikate werden alle auf das erste Argument angewendet, die Funktion gibt `NIL` zurück, wenn mindestens ein Prädikat `NIL` zurückgab. Sind alle Prädikate erfüllt, gibt `all-true?` auch `T` zurück.

[1] Andernfalls erzeugt Screamer eine Fehlermeldung.

```
(defun <csp-funktions-name> (csp-variable constraints)
  (unless (all-true? csp-variable constraints)
          (fail))
  csp-variable)
```

Abbildung 5.1.: Programmiercliché für eine CSP-Funktion

Dies Programmiercliché wurde auch schon im Beispiel Abb. 3.6 verwendet, dort wurde allerdings die CSP-Variable mit ihrer Domain innerhalb der Funktion deklariert. Der CSP-Variable des Programmierclichés in Abb. 5.1 wird außerhalb der Funktion nicht-deterministisch eine Domain zugewiesen (Abb. 5.2). Dazu wird ein nicht-deterministischer Generator eingesetzt, z.B. eine der Screamer-Funktionen `either`, `an-integer-between` oder `a-member-of`.[2] Die Screamer-Funktion `one-value` wandelt einen nicht-deterministischen Ausdruck in einen deterministischen um, indem sie die erste gefundene Lösung zurückgibt.[3]

```
(one-value
  (let ((csp-var <non-deterministic generator>))
    (<csp-funktions-name> csp-var <constraints-list>)))
```

Abbildung 5.2.: Evaluation der CSP-Funktion

Funktionsdefinition durch ein Makros

`defcontain` ist als Makro[4] implementiert, damit die internen Details der Funktionsdefinition vom Komponisten verborgen werden können.

Wie eine Funktionsdefinition innerhalb eines Makros möglich ist, zeigt die Definition eines anderen Arno-Makros, welches ebenfalls das im vorigen Abschnitt vorgestellte Cliché enthält (durch Kursivdruck kenntlich): das allgemeinere, nicht zum Füllen eines CM-Containers vorgesehene und damit deutlich kleinere Makro `defcsp` (Abb. 5.3).[5] Dem Makro werden verschiedene optionale Argumente mit Schlüsselworten übergeben, woraus das Makro die Funktionsdefinition bildet.

Ein Anwendungsbeispiel von `defcsp` sei die Suche nach einem rechtwinkligen Dreieck, dessen Seitenlängen nicht-deterministisch je als Domain deklariert werden (Abb. 5.4)[6]. Die Constraint `gilt-pythagoras?` muß als Prädikat mit nur einem Argument definiert sein, da `all-true?` nur mit solchen Constraints umgehen kann.[7]

Daraus bildet `defcsp` eine Funktionsdefinition. Abb. 5.5 zeigt die erste Stufe einer Makroexpansion von `rechtwinkliges-dreieck`.

[2] Siehe Fußnote 17 auf S. 28.
Eine nicht-deterministische Zuweisung einer Domain, die z.B. alle MIDI-Tonhöhen der kleinen und eingestrichenen Oktave umfaßt, kann mit dem Ausdruck `(an-integer-between 48 72)` erfolgen (siehe Abb. 3.7).

[3] Siehe Abschnitt 3.3.1.

[4] Für eine kurze Einführung in die Makroprogrammierung in Lisp siehe Anhang A.1.

[5] Die in Abb. 5.3 mitgeteilte Definition von `defcsp` ist gekürzt, die vollständige Definition enthält der Anhang C.1.

[6] Die Idee für das Beispiel stammt von Siskind and McAllester [1993, S. 3].

[7] Siehe vorangegangener Abschnitt.

```
(defmacro defcsp (funname &key args let* content-setting fulfil)
  `(defun ,funname (,@args)
      (let* ((result NIL)
             ,@let*)
        (setf result ,content-setting)
        (unless (all-true? result ,fulfil)
            (fail))
        result)))
```

Abbildung 5.3.: Das Makro `defcsp` (gekürzt).

```
(defcsp rechtwinkliges-dreieck
    :args (n)
    :let* ((a (an-integer-between 1 n))
           (b (an-integer-between 1 n))
           (c (an-integer-between 1 n)))
    :content-setting (list a b c)
    :fulfil '(gilt-pythagoras?))

(defun gilt-pythagoras? (seiten)
  (let ((a (first seiten))
        (b (second seiten))
        (c (third seiten)))
    (= (+ (* a a) (* b b)) (* c c))))
```

Abbildung 5.4.: `rechtwinkliges-dreieck` mit `defcsp`.

Die so erzeugte Funktion kann innerhalb von `one-value` aufgerufen werden:

```
? (one-value (rechtwinkliges-dreieck 5))
(3 4 5)
```

Das Beispiel ist länger als das Original [siehe Siskind and McAllester, 1993, S. 3], aber das zugrundeliegende Programmiercliché braucht vom Benutzer nicht gekannt zu werden und separat definierte Constraints sind bei `defcsp` wie bei `defcontain` sehr leicht hinzuzufügen bzw. auszutauschen.

Iteration innerhalb eines nicht-deterministischen Ausdrucks

Um mehr als einen Wert in der (im vorangegangenen Abschnitt) beschriebenen Weise nicht-deterministisch deklarieren und durch Constraints beschränken zu können, wurde der Einsatz einer Schleife gewählt. Eine solche iteriert in `defcontain` durch mehrere CSP-Variablen,[8] die

[8] CSP-Variablen sind in `defcontain` CM-Elemente, deren Slots nicht-deterministisch Werte zugewiesen wurden.

```
(DEFUN RECHTWINKLIGES-DREIECK (N)
  (LET* ((RESULT NIL)
         (A (AN-INTEGER-BETWEEN 1 N))
         (B (AN-INTEGER-BETWEEN 1 N))
         (C (AN-INTEGER-BETWEEN 1 N)))
    (SETF RESULT (LIST A B C))
    (UNLESS (ALL-TRUE? RESULT '(GILT-PYTHAGORAS?)) (FAIL))
    RESULT))
```

Abbildung 5.5.: `rechtwinkliges-dreieck` macroexpanded.

gebunden und deren Bindung anschließend sofort getestet wird. Dabei war zu beachten, daß ein Schleifenzähler bei einem auftretenden Backtracking ebenfalls zurückgesetzt werden muß. Screamer unterstützt dafür neben den üblichen globalen auch *lokale Nebenwirkungen* (local Side Effects).[9] In Screamer werden solche Nebenwirkungen lokal genannt, die mit einem erfolgten BT *zurückgenommen* werden (!). Diese Unterscheidung erlaubt auf der einen Seite die Definition von Funktionen zum Erstellen von Such-Statistiken (globale Nebenwirkung).[10] Andererseits können Zuweisungen zu Lisp-Variablen durch lokale Nebenwirkungen bei einem BT gewissermaßen ungeschehen gemacht werden. Ausdrücke, die lokale Nebenwirkungen enthalten sollen, werden mit dem Screamer-Makro `local` eingeleitet.[11]

Ein Ausdruck wie in Abb. 5.6 evaluiert durch den Einsatz von `local` zu einem Resultat, welches man intuitiv erwartet mag. Eine Schleife, die fünfmal wiederholt wird, fügt in jedem Durchlauf einen nicht-deterministischen Wert mit einer Domain von 1 bis 6 einer Liste hinzu. Dieser Wert soll immer größer sein als vier. Abschließen wird die Liste zurückgegeben.[12]

```
? (one-value
    (let ((liste NIL))
      (local
        (dotimes (i 5)
          (push (an-integer-between 1 6) liste)
          (unless (> (first liste) 4)
            (fail))))
      (reverse liste)))
(5 5 5 5 5)
```

Abbildung 5.6.: Ausdruck mit lokaler Nebenwirkung

[9] Nebenwirkungen sind alles, was *bleibt*, nachdem eine Prozedur, die diese verursacht hat, einen Wert zurückgegeben hat. Dazu gehören z.B. Zuweisungen zu globalen Lisp-Variabeln.

[10] Arno enthält mit der Funktion `print-statistic` eine Funktion, die genau solches tut: die Funktion protokolliert, wie oft jede Constraint versagte (zusammen mit den Angaben, bei wie vielen Variabeln und einer wie großen Domain sie aufgerufen wurde).

[11] Das Gegenstück zu `local` stellt das Screamer-Makro `global` dar, beide können ineinander verschachtelt sein.

[12] Da `push` Elemente am Anfang einer Liste hinzufügt, wird die Liste am Schluß mit `reverse` „umgedreht“. Dies hat bei diesem Beispiel keine Auswirkungen (alle Werte in der Liste sind ja gleich), wohl aber im Beispiel Abb. 5.7.

Dieses Resultat ist aber alles andere als selbstverständlich, wie Abb. 5.7 zeigt. Im Gegensatz zum Beispiel Abb. 5.6 wird in Abb. 5.7 die erfolgte Zuweisung des aktuellen Wertes der Zählervariable `i` zur Variablen `liste` mit `push` nicht mit jedem erfolgten BT zurückgenommen. Es werden alle in der Schleife jemals erfolgten Zuweisungen mitprotokolliert.

```
? (one-value
    (let ((liste NIL))
      (dotimes (i 5)
        (push (an-integer-between 1 6) liste)
        (unless (> (first liste) 4)
          (fail)))
      (reverse liste)))
(1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 1 2 3 4 5)
```

Abbildung 5.7.: Ausdruck wie in Abb. 5.6 aber mit globaler Negenwirkung.

Zu beachten ist, daß zwar Zuweisungen mit `setq`, `setf` oder `push` innerhalb von `local` beim BT zurückgenommen werden. Auch Zuweisungen von CLOS-Slots sind mit `setf` als lokale Nebenwirkung möglich.[13] Für andere Zuweisungsoperatoren, insbesondere auch Slot-Accessors von CLOS-Objekten, gilt dies jedoch nicht. Zudem müssen die Zuweisungen *lexikal* in den Rumpf des `local`-Ausdruck eingebunden sein, dürfen also z.B. nicht in einer eigenen Funktion erfolgen. Jedoch sind Makros (wie `dotimes`), die in den lexikalen Kontext expandiert werden, möglich [vgl. Siskind and McAllester, 1993, S. 5 ff].

Lokale Lisp-Variablen für eine möglichst kompakte Funktionsdefinition

Um die resultierende Funktion möglichst klein zu halten, beginnt `defcontain` mit einem längeren `let*`-Block, in welchem durch Backquotes der Text für die resultierende Funktionsdefinition abhängig von den Nutzerangaben an `defcontain` vorformatiert wird, ähnlich wie im folgenden Pseudo-Beispiel:

```
(defmacro demonstrier-let*-formatierung (name &key arg1 arg2)
  (let* ((arg1-let* (when arg1) <body-arg1>)
         (arg2-let* (when arg2) <body-arg2>))
    `(defun ,name ()
      ,@arg1-let*
      ,@arg2-let*)))
```

Die Definition des Makros `defcontain` ist im Anhang C.1 vollständig aufgeführt.

5.2.2. Konsequenzen aus der nicht vordefinierten Zeitorganisation

Mit Arno kann der Komponist auch die Zeitorganisation einer Partitur deklarieren. Dies kann sowohl die einander folgenden rhythmischen Werte innerhalb eines Thread, als auch, zumindest prinzipiell, die Anzahl der Objekte in einem Thread, die Anzahl der Threads in einem Merge usw. betreffen. Dazu geschieht der Suchvorgang in mehren Schritten.

[13] Für eine Einführung in den Grundgedanken und die Begriffe von CLOS siehe Anhang A.2.

1. Die mit `defcontain` definierte Funktion wird aufgerufen, ihr wird ein CM-Container als Argument übergeben.[14]

2. Innerhalb des übergebenen Containers werden alle Container und Noten so initialisiert, daß das formale Beziehungsnetz (Element x in Container y in Container z) der Deklarierung entspricht.[15] Die Slots aller Elemente werden entsprechend gesetzt oder bleiben ungesetzt. Der `rhythm`-Slot aller Elemente muß mit einem Wert initialisiert sein.

3. Die zeitliche Struktur wird erstmals evaluiert. Es wird die aktuelle Position jedes Objekts in der Zeit festgestellt und die `time`-Slots der Objekte entsprechend gesetzt.

4. Es werden die Slots des ersten Elemente (im ersten Container innerhalb des ersten Containers...) gemäß ihrer Deklaration (mit `:content-setting`) gesetzt.

5. Das aktuelle Element wird getestet, ob es allen (in `:fulfil` und `:avoid` deklarierten) Constraints genügt.

6. Wurden alle Tests erfolgreich absolviert, wird zur nächsten Note vorangeschritten. Zuvor wird wieder die zeitliche Struktur der Partitur aktualisiert. Dieser (etwas zeitaufwendige) Vorgang ist nach jeder Änderung der Partitur erforderlich, damit anschließende Tests sich auch auf die Zeitstruktur beziehen können.[16] Schlug mindestens ein Test fehl, wird ein Backtracking/Backjumping durchgeführt.

7. Wurde die Form nicht-deterministisch deklariert, wird eine Änderung der Form[17] erst vorgenommen, wenn alle Elemente der sich auf Element-Slots beziehenden Domains (Dauer, Tonhöhe...) fehlschlugen.

Zu jedem Zeitpunkt der Suche ist also die aktuelle, bisher gefundene Partitur vollständig in einer in Common Music üblichen Hierarchie von Containern repräsentiert.[18] Für eine weitere Diskussion der Suchstrategie, die durch die nicht vordefinierte Zeitstruktur weniger flexibel und effizient ausfällt, siehe Abschnit 6.2.

5.3. Erweiterungen der Common Music API

Um von einem Objekt aus auf andere Objekte verweisen zu können, waren Funktionen wie `previous-objects` und `simultan-objects` nötig.[19] Diese Funktionen sind in CM nicht definiert,

[14] Enhält diese Funktion weitere mit `defcontain` definierte Funktionen, so muß sie diesen wiederum einen Container übergeben.
Wenn in einer Definition weitere Argumente eingesetzt wurden (in `defcontain` mit dem Keyargument `:further-args`), sind diese natürlich ebenfalls zu übergeben.

[15] Die Deklaration der Form erfolgt in `defcontain` mit den Schlüsselargumenten `:content-type`, `:number` und `:beginning`. Diese Argumente können nicht-deterministisch deklariert werden (Domain für Anzahl der Töne, Domain für Objekt-Typ...). Deren Werte werden in der sequenziellen Suche allerdings erst zuletzt verändert — siehe Punkt 7.

[16] Arno nimmt keinen Test der Constraints daraufhin vor, ob sie sich auf die Zeitstruktur, d.h. den `time`-Slot irgendeines Objektes beziehen. Ein weiteres Argument in `defcontain` „`:always-update-timeslots?`“ wäre denkbar, aber nicht ganz ungefährlich.

[17] Z.B. eine Änderung der Anzahl der Töne, der Anzahl der Container... Siehe Punkt 2.

[18] In einem Multitasking-fähigen Lispcompiler, bei dem mehr ein eine Read-Eval-Loop parallel eingesetzt werden, kann sogar parallel zur Suche der aktuelle Zustand der Partitur abgefragt und damit der Fortgang der Suche beobachtet werden.

[19] Siehe Abschnitt 70.

sie werden aber innerhalb einer zwei- oder mehrstelligen Constraintsdefinition (die ja jeweils nur ein Objekt als Argument erwartet)[20] gebraucht. Auch zur Deklarierung einer Domain, die von Parametern anderer Objekte abhängt, sind solche Funktionen nötig.[21] Mit Hilfe der gut dokumentierten CM-API[22] [Taube, b] und den Mitteln der objektorientierten Erweiterung von Common Lisp, CLOS [Keene, 1989],[23] war die Definition von Erweiterungen der CM-API leicht möglich. Beispielhaft wird im folgenden an Hand von `previous-objects` und `simultan-objects` skizziert, wie solcher Funktionen implementiert wurden.

Die CM-Funktion `object-position` ermittelt die Position eines Objektes innerhalb eines Containers. Mit der CM-Funktion `nth-object`, die das Objekt an Position n im übergebenen Containter zurückgibt, und der Position eines Objektes können alle Objekte vor der Position des fraglichen Objektes erreicht werden. Die genaue Implementierung von `previous-objects` findet sich im Anhang C.2.

Mit Hilfe der Slots `time`[24] und `duration`[25] läßt sich das Zeitintervall (Startzeit – Endzeit) jedes Objektes bestimmen. Durch einen Vergleich des Zeitinterfalls eines Objektes mit dem Zeitintervall aller anderen Objekte lassen sich die simultanen Objekte herausfiltern. Die genaue Implementierung von `simultan-objects` findet sich im Anhang C.2.

Um den Umgang mit CM-Objekten für den Einsatz in Arno einfacher zu gestalten, wurden eine Reihe ähnlicher Funktionen definiert, deren Implementierung sich ebenfalls im Anhang C.2 findet.

[20] Siehe Abschnitt 4.3.

[21] In `defcontain` kann z.B. in `:content-setting` die Tonhöhe von MIDI-Noten als Intervall abhängig von ihrer jeweiligen Vorgängernote deklariert werden.

[22] Siehe Abschnitt 3.2.2.

[23] Siehe Anhang A.2.

[24] Die Einsatzzeit eines Objektes. Dieser Slot wird durch Aufruf der CM-Funktion `run-object` mit einem Container für alle Objekte innehalb dieses Containers abhängig vor ihren rhythmischen Werten und eventueller Offsets gesetzt, siehe Abschnitt 3.2.2.
Jedes CM-Element (Note, Pause...) und auch jeder Container enthält diesen Slot, siehe Abschnitt 3.2.1.

[25] Die Dauer des jeweiligen Objektes. Ist dieser Slot leer, wird `rhythm`, die Zeit bis zum nächsten Object, ausgewertet.

6. Offene Fragen

6.1. Herausforderung: Komponieren durch Beschreiben

Die eigentliche Motivation, ARNO zu programmieren, war es, ein Komponieren durch das Beschreiben des gesuchten Ergebnisses mit Hilfe von Constraints zu ermöglichen. Dieses Beschreiben erweist sich in der konkreten Anwendung als gar nicht so einfach. Es bedarf einer ziemlich großen Abstraktion, ein musikalisches Resultat durch Regeln/Constraints zu definieren. Dies Problem war prinzipiell erwartet worden, viele Autoren weisen auf diese Schwierigkeit hin [Cope, 1991; Ebcioğlu, 1992]. Da ARNO aber gerade auch mit nicht traditionelle Kompositionsregeln arbeiten kann, taucht hier zusätzlich die Schwierigkeit auf, daß ein in traditioneller Satztechnik geschulter Komponist stark durch diese traditionellen Regeln geprägt ist. Für einen solchen Komponisten (wie auch den Autor) ist es erstaunlich schwer, neue und sinnvolle Kompositionsregeln zu formulieren — doch gerade das bleibt kompositorische Herausforderung.

Bislang wurde ARNO lediglich peripher kompositorisch eingesetzt. Die Komposition *Klavierkreisel* (1999) des Autors enthält eine Folge von Akkorden, die erst als Glissando erklingen, das von Akkord zu Akkord immer schneller wird, bis die Töne in einen einzigen Anschlag zusammenfallen, die einem Glockenschlag ähnelt. Die Harmonik der Akkorde ist mikrotonal. Diese Akkordfolge wurde mit Hilfe von ARNO komponiert, es sind Constraints zur Harmonik und zur Stimmführung deklariert worden. Das Beispiel demonstriert, daß es praktikabel sein kann, nicht ganze Stücke, sondern nur Abschnitte zu programmier-komponieren.

6.2. Erweiterungen zur Effizienzsteigerung

Die derzeitige Implementierung der Suche in Arno ist nicht sonderlich effizient, insbesondere, wenn Constraints eingesetzt werden, die sich auf simultane Events beziehen. Es erweist sich für die Suchstrategie als grundsätzlich problematisch, daß auch die Zeitstruktur der resultierenden Partitur durch die Suche gefunden werden soll. Dadurch stellt sich erst während der Suche heraus, welche Events zueinander simultan sind. Diese Zuordnung kann sich zudem während der Suche ändern. Im folgenden werden einige alternative Suchstrategien mit Blick auf ihre Verwendung in einem Programm wie ARNO diskutiert.

6.2.1. Bearbeitung der Domain vor der Suche

Zwei Strategien zur Performance- aber auch zur Qualitätssteigerung sind (wenn auch nicht sonderlich elegant) bereits in der vorliegenden Version von Arno einsetzbar: beide lassen sich als Funktionen definieren, welche die deklarierte Domain verändern.

Einstellige Constraints, d.h. Constraints die sich auf nur einen Parameter eines einzelnen Elements beziehen, lassen sich als Filter für die Domain dieses Parameters formulieren. Sollen beispielsweise alle Tonhöhen der Elemente eines Containers einer bestimmten Skala angehören, kann ein Filtern der Tonhöhen-Domain eine Performancesteigerung erreichen. Es ist im Beispiel Abb. 4.7 Knoten-Konsistenz[1] dadurch erreicht, daß durch die Funktion `mode-degree` alle Werte der Tonhöhen-Domain der ebenfall im Beispiel definierten Skala `c-major` angehören.

Auf ähnliche Weise lassen sich auch *Heuristiken* für die Suche formulieren: eine Funktion ändert die Reihenfolge der Werte innerhalb der Domain so, daß wahrscheinlich erfolgreiche oder auch für das Ergebnis bessere Werte in der Domain weiter vorn liegen. Eine Zufallsanordnung mit `a-shuffled-expr` wie im Beispiel Abb. 4.1 ist eine einfache Heuristik.

Eine etwas raffiniertere Heuristik ist zu klassischen Stimmführungsregeln vorstellbar. Die Definition der Domain eines Elements in `defcontain` kann auch abhängen von den Slot-Werten bereits instanziierter Elemente. Die Tonhöhendomain kann z.B. erlaubte Intervalle (die zur Vorgängernote addiert werden) an Stelle von Tonhöhen enthalten. Diese sind der Größe nach aufsteigend geordnet, dadurch sind Schritte und kleine Intervalle warscheinlicher als große Sprünge. Gesondert ist nur die Tonhöhe der allererste Note zu behandeln [vgl. Ebcioğlu, 1992, S. 320].

Unter Einsatz eines Filters (wie der LISP-Primitive `remove-if-not`), der eine Domain z.B. abhängig von der jeweiligen Vorgängernote und Constraints reduziert, ist auch Forward Checking möglich.[2]

Mit diesen Ansätzen würde aber (wie im Beispiel Abb. 4.7 durch `mode-degree` oder im Beispiel Abb. 4.1 durch `a-shuffled-expr)` eine Domain „unmittelbar“ bearbeitet werden, indem eine Liste verändert wird, die einem Ausdruck wie `a-member-of` übergeben wird. Daraus resultiert eine weniger übersichtliche Deklaration. Auch sind die Filterfunktionen innerhalb der Domaindeklaration in der derzeitigen Implementation nicht leicht austauschbar.

Eine Funktion, die beliebig viele einstellige Constraints oder Heuristiken in die Deklaration einer bestimmten Parameterdomain einbindet, und damit ein Austauschen ermöglicht, ist leicht zu ergänzen.

Ein alternativer — für den Komponisten einfacherer — Ansatz mit weiteren Schlüssel-Argumenten für `defcontain` (zusätzlich zu den Argumenten wie `:content-setting`, `:fulfil`, `:avoid` usw. je eins für einstellige Constraints und Heuristiken) wäre weniger leicht realisierbar. Sowohl Heuristiken wie auch einstellige Constraints müssen sich auf beliebige Parameter (Dauer, Tonhöhe...) beziehen können. Das Makro `defcontain` müßte testen, auf welchen Parameter sich ein Ausdruck bezieht, um ihn in der resultierenden Funktion dort einzusetzen, wo die Domain dieses Parameters deklariert wird.

6.2.2. Reihenfolge der CSP-Variablen

In der derzeitigen Implementierung von ARNO steht die Suchreihenfolge aller CSP-Variablen (d.h. COMMON MUSIC-Elemente)[3] mit Beginn der Suche fest: ihre Reihenfolge ergibt sich aus ihrer Position in einem Container, aus dessen Position in einem übergeordneten Container, aus

[1] Siehe Abschnitt 2.3.2.
[2] Siehe Abschnitt 2.3.3.
[3] Daß jedes Element genaugenommen mehrere CSP-Variablen enthält, je eine pro in die Deklaration einbezogenen Slot, ist für die Überlegung hier nicht wichtig.

der Position des übergeordneten in einem überübergeordneten Container usf. Der Algorithmus beginnt die Suche im ersten Element im ersten Container im ersten Container... und setzt sie mit dem zweiten Element im ersten Container im ersten Container fort. Der Algorithmus arbeitet also von unten nach oben zuerst alle Elemente eines einstimmigen Threads vollständig ab und schreitet erst dann zum nächsten Thread fort. Eventuell auftretende Unstimmigkeiten zu simultanen Elementen in anderen parallelen Threads werden deshalb erst spät bemerkt.

Der daraus resultierende Performanceverlust läßt sich mit keiner anderen statischen Anordnung der CSP-Variablen (siehe Abschnitt 2.3.4) prinzipiell verhindern. Da wie schon erwähnt nicht bereits vor der Suche feststeht, welche Elemente in der resultierenden Partitur gleichzeitig sein werden — auch dies soll ja erst durch die Suche ermittelt werden — ist keine im voraus definierte Suchreihenfolge möglich, die nacheinander erst alle simultanen Elemente abarbeitet, um dann weiter in der Partitur voranzuschreiten.

Eine Verbesserung der Performance ist durch eine dynamische Reihenfolge der CSP-Variablen vorstellbar: ein Algorithmus bestimmt vor jedem Voranschreiten bei der Suche die als nächstes zu bindende Variable abhängig vom aktuellen Zustand der Partitur und setzt die Suche direkt mit dieser Variablen fort. Dadurch läßt sich eine parallele Abarbeitung zeitlich paralleler Container realisieren, auftretende Unstimmigkeiten zu Constraints, welche die simultanen Elemente einschränken, werden sofort bemerkt.

Ein solcher Algorithmus müßte vor jedem Voranschreiten zum nächsten Element alle Elemente (in den Sub- und Subsubcontainern) sortieren nach ihrem `time`-Slot und ihrer Threadposition. Allerdings müßte der Algorithmus die Deklaration der Domain und die Liste der zu erfüllenden Constraints für jedes Element zusammen mit diesem (durch ein Makro?) kapseln, damit die Deklaration einer Domain abhängen kann von Slots anderer Elemente, die zum Zeitpunkt der Initialisierung noch nicht gebunden sind. Das Sortieren müsste jedoch vor jedem Voranschreiten zum nächsten Event in der Partitur erfolgen, der erhoffte Performancegewinn ist damit vom konkreten kompositorischen Problem abhängig.

Backjumping (BJ) in Arno

In der gegenwärtigen Fassung von Arno ist ein Backjumping-Algorithmus implementiert, allerdings ist dieser nur bei der Deklaration eines Slots pro Element sinnvoll einsetzbar. Dieser wurde entwickelt, um dem Performanceverlust durch die Reihenfolge der Variablen während der Suche in der derzeitigen Arno-Version zu begegnen (siehe oben). Durch ein BJ wird verhindert, daß nach einer schließlich bemerkten Unstimmigkeit zwischen simultanen Elementen durch BT zunächst „unbeteiligte“ Elemente korrigiert werden, die in der Partitur erst zeitlich nach dem Konflikt auftreten. Durch BJ springt die Suche direkt zum Konfliktverursacher, was häufig eine erhebliche Performancesteigerung zur Folge hat.

Backjumping kann in Arno durch Setzen des Flags `:bj`?, ein Argument von `defcontain`, zusammen mit dem Einsatz spezieller Arno-Varianten nondeterministischer Screamer-Funktionen, nämlich `either-bj`, `an-integer-beween-bj` und `a-member-of-bj` erreicht werden (siehe Abb. 6.1). Die Implementation dieser Varianten ist im Anhang C.3 zu finden.

```
(defcontain <name>
  :bj? T
  <...>
  :content-setting
  (set-object (current-object)
              '<slot> (<nondeterministic-generator>-bj <params>)
              <...>))
```

Abbildung 6.1.: Schablone für den Einsatz von Backjumping in ARNO

6.2.3. Konsistenzkontrollen

Es ist wiederum der nicht vordefinierten Zeitstruktur geschuldet, daß Konsistenz-Techniken[4] in ARNO schwierig bis nicht einsetzbar sind. Bei einer Partitur-Repräsentation mit *vordefinierter* Zeitstruktur ließen sich alle mehrstelligen Constrains in binäre Constraints überführen. In diesem Fall könnte mit Techniken der Kanten-Konsistenz die Domain reduziert werden, so daß in einer anschließenden Suche BT zumindest weniger häufig nötig wäre.

Im Falle einer vordefinierten Zeitstruktur wäre also folgendes Modell vorstellbar: Ausgangspunkt sei, daß sich für jedes Element der Partitur Constraints (außer auf das Element selber) nur beziehen dürfen auf beliebig viele Vorgänger in derselben Stimme oder auf simultane Elemente. Wenn nun alle Vorgänger und alle simultanen Elemente eines jeden Elements in je zwei zusätzlichen gekapselten CSP-Variablen zusammengefaßt werden, ließen sich alle Constraints als binäre Constraints ausdrücken [siehe Bartak, 1998, binary.html]. Das resultierende Netz könnte durch eine Suche mit Konsistenzkontrollen effizienter gelöst werden [siehe Bartak, 1998, propagation.html].

Bei einer nicht vordefinierten Zeitstruktur allerdings erscheint ein solches Vorgehen wenig praktikabel: es müßte davon ausgegangen werden, daß alle nicht der jeweiligen Stimme angehörenden Elemente potenziell simultane Elemente seien. Eine Einschränkung des Suchraums käme dann lediglich einer Vorordnung der Suche gleich: „Falls *dieses* Element diese Dauer und alle seine Vorgänger jene jeweilige Dauern haben sollten, und *jenes* Element und seine Vorgänger solche Dauern, dann wären beide *simultane* Elemente, woraus folgende Einschränkung der Domain resultieren würde...“ Ein solch umfangreiches Vortesten würde die Performance wahrscheinlich negativ beeinflussen, von den Schwierigkeiten der Implementierung mal abgesehen.

6.3. Weitere Erweiterungsmöglichkeiten

6.3.1. Generierung oder Variation

In der vorliegenden Version werden in ARNO Container jedesmal *neu* gefüllt, aller eventuell zuvor enthaltener Inhalt des Containers wird verworfen. Eine frühere Programmversion generierte nicht jedesmal von Grund auf neu, sondern *variierte* den Inhalt des Containers gemäß der Deklaration, d.h. änderte nur bestimmte Slots bestimmter enthaltener Elemente. Beide Ansätze allein sind unbefriedigend.

Eine jedesmal neu erfolgende Generierung erfordert jedesmal die ganze Rechendauer. Auch wenn Teile des Ergebnisses nicht verändert sein werden, werden sie jedesmal neu berechnet. Ein rein

[4] Siehe Abschnitt 2.3.2.

variierender Ansatz dagegen ist für den Komponisten schwerer zu handhaben. Es muß zunächst ein zu variierender Containerinhalt vorliegen. Dieser determiniert z.B. die Form des Resultats, vielleicht auch die Zeitstruktur (so in PWConstraints, vgl. Abschnitt 2.4.4). Sollen dagegen nur einzelne Elemente oder Container unverändert bleiben bzw. variiert werden, müssen diese entsprechend gekennzeichnet werden. Eine Kombination beider Ansätze ist zu bedenken.

6.3.2. Gewichtung von Constraints

Musikalische Regeln sind bestenfalls in Lehrbüchern absolut; ja, selbst in Lehrbüchern kommt es vor, daß sich einzelne Regeln gegenseitig ausschließen, daß also zwischen wichtigen und weniger wichtigen Regeln unterschieden werden muß.[5] In der derzeitigen Fassung kann ARNO nur mit *absoluten* Constraints umgehen, alle Constraints werden unbedingt eingehalten. Die Möglichkeit, Constraints zu gewichten, zwischen mehr und weniger wichtigen Constraints zu unterscheiden, ist aber sehr wünschenswert.

Das Problem, daß sich einzelnen Constraints gegenseitig ausschließen, tritt natürlich nicht erst mit der musikalischen Anwendung auf. Es sind verschiedene Ansätze entwickelt worden, *Over-Constraint Problems* zu handhaben [vgl. Bartak, 1998, Over-Constraint Problems]. Auch lokale Suchstrategien, Algorithmen, die nach lokalen Optima suchen, verknüpft mit stochastischen Heuristiken, können ein hilfreicher Ansatz sein, die Beschränkungen zu überwinden, die sich aus der erschöpfenden Suche der systematischen Suchalgorithmen ergeben [vgl. Bartak, 1998, Heuristics and Stochastic Algorithms].

6.3.3. Veränderliche Constraints

Musik lebt häufig davon, daß sie sich im Ablauf der Zeit entwickelt. So ist es auch vorstellbar, daß sich ein Satz von Constraints über die Dauer eines Stückes ändert, entwickelt. David Copes Komposition *Mozart in Bali* [Cope, 1991, S. 216, 220ff] demonstriert diesen Gedanken — wenn auch mit einer anderen Technik realisiert — sehr deutlich. Eine Musik, die der Stilistik Mozarts nahe steht, geht in einer langsamen Entwicklung über in eine Musik im Balinesischen Gamelan-Musik-Stil.

In der Constraintsliteratur war vom Autor bislang kein Hinweis auf Ansätze zu Interpolationen zwischen Constraints in der Constraints-Programmierung gefunden worden. Veränderliche Constraints aber sind in ARNO schon jetzt mit Hilfe von Hüllkurven innerhalb einer Constraintdefinition implementierbar, wie im Beispiel Abb. 4.15 für den Rhythmus geschehen. Diese Idee ließe sich ausbauen.

Auch sich allmählich verändernde Heuristiken, die die Domain einzelner musikalischer Parameter bearbeiten, sind vorstellbar und könnten z.B. mit einer Hüllkurve gesteuert werden.

[5] „Sollen also diese Sätzchen, wie alles, was ein Kunst- und Formgefühl entwickeln soll, schon *Stil* haben, das heißt ein gewisses ebenmäßiges Verhältnis zwischen Wirkung und aufgewendeten Mitteln zeigen, so wird es wohl zweckmäßig sein, die Anweisungen der alten Harmonielehre zu berücksichtigen. Wir werden sie beruhigt fallen lassen dürfen, sowie unsere Mittel reicher, unsere Möglichkeiten, dem Material Wirkungen abzuringen, also größer werden. Wir nehmen auch diese Anweisungen nicht als Regel, das heißt als etwas, das nur durch eine Ausnahme aufgehoben werden kann, sondern eben nur als Anweisungen. Und wir sind uns klar darüber, daß wir sie nur deswegen berücksichtigen, weil wir stilistisch richtige Wirkungen nur erzielen können, wenn wir die Mittel ihren einfachen Zwecken angemessen verwenden.“ [Schönberg, 1911, S. 48]

7. Fazit

Das computergestützte Komponieren eröffnet radikal neue kompositorische Möglichkeiten, zeigt aber auch Beschränkungen, die wiederum vom gewählten Programmierparadigma abhängen. Die weit verbreitere prozedurale Programmierung etwa ermöglicht und erzwingt neue kompositorische Arbeitsweisen und musikalische Stile, sie erschwert insbesondere die gleichzeitigt Beachtung verschiedener kompositorischer Aspekte wie Harmonik, Stimmführung und Instrumentierung — um ein paar traditionelle Aspekte zu nennen.

Durch den Autor wurde ein anderes Programmierparadigma, das Programmieren mit Constraints, für die Entwicklung von Arno, einer Erweiterung der Kompositionsumgebung Common Music, eingesetzt. Arno erlaubt dem Komponisten eine andere Arbeitsweise sowohl im Vergleich zum computergestützen prozeduralen „Programmier-Komponieren“ als auch im Vergleich zum herkömmlichen Komponieren „mit Papier und Bleistift“: der Komponist deklariert Constraints — Beziehungen, die zwischen musikalischen Ereignissen und ihren Parametern eingehalten werden sollen. Constraints können sich dabei auf alle musikalischen Parameter (z.B. die Zeit, die Tonhöhe, die Dynamik und beliebige weitere Klangsyntheseparameter) beziehen und beliebige Elemente der Partitur (z.B. eine oder mehrere einander folgende Noten oder alle gleichzeitig klingenden Noten) können untereinander durch Constraints verbunden werden. Das Programm erstellt darauf ein Ergebnis, das alle geforderten Bedingungen erfüllt.

Diese Arbeitsweise ist einerseits dem herkömmlichen Komponieren näher als die computergestützte Komposition mit einer prozeduralen Programmiertechnik, andererseits verlangt sie vom Komponisten eine ungewohnt umfassende Abstraktion von einer konkreten musikalischen Gestalt. Dieses verleitet zumal den traditionell geschulten Komponisten dazu, zunächst klassische Tonsatzregeln z.B. des Kontrapunkts zu implementieren. Strenge musiktheoretische Lehrwerke, die häufig viel mehr einer Schulung des musikalischen Überblicks, Stilgefühls und Gehörs dienen und damit zu einer zunehmenden kompositorischen Freiheit erziehen wollen, statt gültige Kompositionsgesetze zu verkünden, werden eingesetzt, um eben genau solche Kompositionsgesetze zu programmieren. Ist dies jedoch lediglich ein Stadium, in dem der Komponist sich die neue Arbeits- und Denkweise aneignet, die die kompositorische Nutzung des Constraintsparadigmas erlaubt und erzwingt, darf man auf zukünftige musikalische Ergebnisse gespannt sein.

Offenbar gilt immernoch, was Herbert Brün schon 1969 beobachtete: Computer-Music-Systeme sind nicht neutral, sondern beschränken den Komponisten durch die jeweils von ihnen zur Verfügung gestellte Menge von Operationen. Jeder Blickwinkel auf ein Stück (sei er durch eine bestimmte Kompositionsstrategie oder Partiturrepräsentation bestimmt) wirkt als ein Filter, der die Aufmerksamkeit des Betrachters in eine bestimmte Perspektive lenkt. Ein Bemühen um den „Heilige Gral einer universalen Repräsentation“ stellt gegenüber einer kreativen Musikaufassung sogar einen Gegensatz dar [vgl. Roads, 1996, S. 856].

Die Arbeit an Arno ist mit der Abgabe der Diplomarbeit nicht abgeschlossen. Es kann die Effizienz noch gesteigert und auch der Leistungsumfang erweitert werden. Die wichtigste Erweiterung der nächsten Zeit ist die Änderung der Suchreihenfolge, in der die einzelnen Elemente/Noten besucht werden, um die Effizienz zu verbessern. In der derzeitigen Version werden — traditionell ausgedrückt — die einzelnen Stimmen nacheinander erzeugt, erst wird eine Stimme abgeschlossen, bevor zur nächsten vorangeschritten wird. Eventuelle Verstöße gegen Constraints zu simultanen Tönen werden dadurch erst spät bemerkt und das Programm muß deshalb viel überflüssige Arbeit leisten. Ein Ansatz, wie die Suchreihenfolge geändernt werden kann, wird im Text vorgeschlagen. Eine wichtige Erweiterung des Leistungsumfangs wäre die Einführung gewichteter Constraits, durch die die Wichtigkeit verschiedener Constaints graduell skalierbar wäre.

Anhang

A. Lisptechniken

A.1. Makros in Lisp

Wie in anderen Programmiersprachen ist auch in Lisp das Definieren von Funktionen bzw. Prozeduren möglich und eine zentrale Technik. Die dadurch erreichte Abstraktion erlaubt das Verbergen von Details und gestattet eigentlich erst, größere Programme zu realisieren. Bei der Ausführung steht jedoch die Reihenfolge der Berechnungen fest. Will der Programmierer diese ändern, also z.B. neue Kontrollstrukturen definieren, kann er dazu in Lisp Makros schreiben. Makros sind — wie auch in anderen Programmiersprachen — ein Mittel zur textuellen Programmtransformation, Lisp-Makros sind jedoch besonders ausdrucksstark.

Beispielsweise gibt es in Common Lisp nicht die in vielen anderen Programmiersprachen bekannte Schleife `while`. Der Programmierer kann dies ändern, indem er ein Makro definiert, daß einen Ausdruck der Form

```
(let ((i 0))
  (while (< i 5)
    <weitere Ausdrücke>
    (setf i (1+ i))))
```

umformt in eine Lisp bekannte Schleife. Ein solches Makro könnte beispielsweise die sehr allgemeine Schleife `do` einsetzen [Graham, 1994, S. 91]:

```
(defmacro while (test &body body)[1]
  `(do ()
       ((not ,test))
     ,@body))
```

Die zunächst etwas befremdliche Notation mit Kommas und Backquote (`) stellt eine schreibtechnische Abkürzung dar: dadurch werden die Ausdrücke, die beim Aufruf von `while` den Argumenten `test` und `body` übergeben werden, an der entsprechenden Stelle in der `do`-Schleife eingesetzt.[2] Wohlgemerkt: nicht die Ergebnisse von `test` bzw. `body` werden übergeben, wie dies

[1] Das Schlüsselwort &`body` geht einem Argument `body` voran, dem mehrere Ausdrücke zugewiesen werden können, die in einer Liste zusammengefaßt werden. Auf diese Weise kann der „Rumpf“ für `while` übergeben werden.

[2] Das Backquote leitet einen Ausdruck ein, der nicht evaluiert werden soll. Mit einem Komma eingeleitete Ausdrücke innerhalb eines Ausdrucks mit vorangestelltem Backquote werden dagegen dennoch evaluiert, ihr Wert wird an ihrer Stelle eingesetzt. Eine Sonderform sind Symbole mit vorangestelltem `,@`, die zu einer Liste evaluieren. Die Ausdrücke in der Liste werden an dieser Stelle ohne die sie zusammenfassende Liste eingefügt.

in einer Funktion geschehen würde, sondern quasi der Programmtext, der den Argumenten übergeben wurde. Mit anderen Worten: mit einem Makro läßt sich ein Programm definieren, daß ein anderes Programm (hier die `do`-Schleife) zurückgibt: „A function produces results, but a macro produces *expressions* — which, when evaluated, produce results.“ [Graham, 1994, S. 82]. Die durch ein Makro vorgenommene Umformung des Programmtextes kann durch den Programmierer mit einer *Makroexpansion* kontrolliert werden. Ein Ausdruck, der „ganz außen“ ein Makro enthält, kann mit einer Funktion zur Makroexpansion evaluiert werden zu dem Ausdruck, den das Makro erzeugt (Abb. A.1). Dies ist zum Debuggen von Makros sehr hilfreich.

```
? (macroexpand-1
    '(while (< i 5)
      (print i)
      (setf i (1+ i))))
(DO ()
    ((NOT (< I 5)))
  (PRINT I)
  (SETF I (1+ I)))
T
```

Abbildung A.1.: Makroexpansion

In einem Makro kann auch eine Funktionsdefinition mit `defun` enhalten sein, ein Makro kann somit eine Funktionsdefinition vorformatieren.[3]

Mit Makros läßt sich die Evaluation von Lispausdrücken kontrollieren. Bei der Compilierung eines Programms wandelt der Parser mit Makros formulierte Ausdrücke in deren resultierende Lispausdrücke um, bevor das Programm an den Compilier weitergereicht wird. Mit Makros kann der Programmierer also in diesem Zwischenstatus ein Programm manipulieren [vgl. Graham, 1994, S. 83].

Ausführlich werden Makros von Graham [1994] erläutert.

A.2. CLOS, objektorientierte Programmierung mit Common Lisp

Das Common Lisp Object System (CLOS, [Keene, 1989]) ist eine standardisierte Erweiterung von Common Lisp für das objektorientierte Programmierparadigma. Gegenüber dem prozeduralen Programmierparadigma bestehen die Vorteile der Objektorientierung vor allem darin, daß größere Programmierprojekte leichter überschaubar sind, leichter erweitert werden können und das eine Wiederverwendung von Code in anderen Projekten einfacher ist.

Die leichtere Überschaubarkeit wird durch ein Modell erreicht, das unserer Welterfahrung näher ist, als die rein prozedurale Denkweise. Grundelemente der objektorientierten Programmierung sind nicht verschiedene Datentypen, an denen verschiedene Prozeduren vollzogen werden. Grundelement ist das Objekt, in dem sowohl Daten als auch Programme gekapselt sind.

Das Objekt `meinFahrrad` beispielsweise enthalte u.a. Angaben über seine Farbe und seine Größe. In CLOS wird eine im jeweiligen Objekt enthaltenen Variable, die solche Attribute enthält, *Slot*

[3] Siehe S. 50.

genannt. Deren Wert ist über einen *Slot-Accessor* zugänglich. Das Objekt `meinFahrrad` ist aber nur eine *Instanz* der *Klasse* `Fahrrad`. Die Klasse umfaßt u.a. die Angaben, welche Eigenschaften in einem Objekt gespeichert werden können. Verschiedene inhaltlich zusammengehörende Daten lassen sich damit als Einheit repräsentieren.

Für eine Klasse können *Methoden* definiert werden, Funktionen die nur diese Klasse auswerten kann. Durch dieses Konzept können Objekte verschiedener Klassen auf denselben Methodennamen verschieden reagieren. Es ist möglich, daß ein allgemeiner Methodenname — etwas `bremse` — von verschiedenen Objekten verschiedener Klassen jeweils angemessen umgesetzt wird. Der Bremsmechanismus einer Instanz der Klasse `Auto` etwa kann anders funktionieren als der einer Instanz der Klasse `Fahrrad`, aber der Programmierer muß sich nur einen Befehl merken.

Schließlich ist durch das Konzept der *Vererbung* ein Programm aber auch leichter erweiterbar, als eine rein prozedurale Programmierung erlauben würde. Eine Kindklasse ererbt von ihrer Elternklasse alle Slots und Methoden. Für eine Kindklasse können neue Slots und Methoden definiert werden. Diese sind nur in der Kindklasse und deren Kindklassen enthalten. Es ist zweckmäßig, daß die Kindklasse stärker spezialisiert als die Elternklasse ist. Beispielsweise könnten die Klassen `Mountainbike` und `Tandem` Spezialisierungen der Klasse `Fahrrad` darstellen.

Es können Methoden einer Elternklasse überschrieben werden. Grundsätzlich verdeckt dabei die Methode einer Kindklasse die Methode der Elternklasse mit dem gleichen Namen — eine Methode der im Vergleich zur Elternklasse stärker spezialisierten Kindklasse wird ebenfalls spezialisierter als die gleichnamige Methode der Elternklasse sein.

CLOS erlaubt *Mehrfachvererbung*: eine Klasse kann mehr als eine Elternklasse haben.

Objektorientierte Programmierung hilft für eine bessere Übersicht und und leichtere Pflege von Programmen. Weil die Definition einer einzelnen Objektklasse gut vom übrigen Programm gekapselt ist, läßt sie sich gut extrahieren und in anderen Programmen wiederverwenden.

A.3. Nondeterminismus in Lisp

Nondeterminismus läßt sich in Lisp recht elegant durch *Continuations* realisieren. Dieser Mechanismus wird (anders als beim Lisp-Dialekt Common Lisp) vom Lisp-Dialekt Scheme[4] bereits vom Sprachstandard gut unterstützt. Der folgende Abschnitt setzt deshalb Scheme ein, obwohl sich Continuations auch in Common Lisp implementieren lassen [5]. Dieser Abschnitt referiert Graham [1994, S. 258ff].

A.3.1. Continuations

Eine Continuation ist eine Funktion, die eine zukünftige Berechnung repräsentiert. Es läßt sich mit ihrer Hilfe der aktuelle Zustand einer Berechnung quasi einfrieren, um später die Berechnung genau am festgehaltenen Punkt fortzusetzen.

Für diesen — zugegebenermaßen zunächst schwer zu verstehenden — Zweck stellt Scheme den Operator `call-with-current-continuation` zur Verfügung, der in der folgenden Form aufgerufen werden kann:

[4] Für die wichtigsten Unterschiede der Lisp-Dialekte Common Lisp und Scheme siehe Graham [1994, S. 259].
[5] Siehe Graham [1994, S. 266ff]

```
(call-with-current-continuation
  (lambda (cc)
    <...>))
```

Der Operator übergibt einer (mit `lambda` definierten) anonymen Funktion die Continuation (eine weitere Funktion, hier `cc` zugewiesen), die die zukünftige Berechnung ab dem Aufruf von `call-with-current-continuation` repräsentiert. Die Continuation kann durch Zuweisung in einer LISP-Variablen gespeichert werden.[6] Wird die Continuation angewandt, so erwartet sie als einziges Argument die zuvor zu erfolgenden Berechnungen, die dann sozusagen textuell innerhalb von `call-with-current-continuation` stehen. Das Beispiel in Abb. A.2 [vgl. Graham, 1994, S. 261] wird dies besser verdeutlichen.

```
;; eine globale Scheme-Variabele muß vor Zuweisung
;; mit set! zunächst initialisiert werden
? (define eingefroren ())
eingefroren

;; Aufruf einer Continuation
? (append '(eingefroren gab zurueck:)
          (list (call-with-current-continuation
                  (lambda (cc)
                    (set! eingefroren cc)
                    'x))))
(eingefroren gab zurueck: x)
```

Abbildung A.2.: Zuweisung einer Continuation in SCHEME

Der Variablen `eingefroren` wurde durch eine Continuation sozusagen alles das zugewiesen, was textuell außerhalb des Aufrufs von `call-with-current-continuation` steht, sie ist vergleichbar mit der Funktion:

```
(lambda (berechnung-vorher)
  (append '(eingefroren gab zurueck:)
          (list berechnung-vorher)))
```

Die zuvor zu erfolgende Berechnung (zunächst lediglich `'x`) muß dieser Continuation bei einem späteren Aufruf übergeben werden:

[6] Im Gegensatz zu COMMON LISP unterscheidet SCHEME nicht zwischen dem Symbolwert und dem Funktionswert eines Symbols. Mit dem Operatoren `define` und `set!` werden deshalb sowohl Funktionswerte als auch Symbolwerte zugewiesen. Wurde einer Variablen eine Funktion zugeweisen, so kann diese Funktion direkt (ohne `funcall` oder `apply` wie in COMMON LISP) aufgerufen werden:

```
? (define add (lambda (x y) (+ x y)))
add
? (add 1 2)
3
```

```
? (eingefroren 'nochmal)
(eingefroren gab zurueck: nochmal)
```

Aber eine Continuation ist nicht lediglich eine Funktion (einschließlich aller lexikalischen Bindungen), sondern enthält eine Kopie des Kellers (*Stack*) zum Zeitpunkt ihrer Erzeugung. Zum Zeitpunkt der Zuweisung von `eingefroren` folgte `(append ...)` unmittelbar der `top-level`. Deshalb erzeugt der folgende Ausdruck keinen Fehler, was er bei einer Funktion getan hätte. Die Berechnung „außerhalb" der Continuation, die diese aktiviert, wird ignoriert:

```
? (1+ (eingefroren 'kein-problem))
(eingefroren gab zurueck: kein-problem)
```

A.3.2. Nondeterminismus mit Continuations

Mit diesem Mittel kann ein Funktionspaar `choose` und `fail` definiert werden, welches sich recht intuitiv handhaben läßt und sich sehr ähnlich wie `either` und `fail` in SCREAMER verhält.[7] Die Funktion `choose` wählt nicht-deterministisch[8] einen Wert aus einer ihr übergebenen Liste, bestimmte Werte werden später durch die Funktion `fail` untersagt.

```
? (let ((x (choose '(1 2 3 4 5))))
    (if (not (< 3 x))
        (fail))
      x)
4
```

Wie in Abb. A.3 [vgl. Graham, 1994, S. 292] zu sehen ist, kann mit Continuations sehr konzentriert die Simulation von nicht-deterministischem Programmieren durch BT realisiert werden. Der Funktion `choose` wird eine Liste von Alternativen (`choices`) zugewiesen. Ist diese Liste leer, wird `fail` aufgerufen. Andernfalls wird der globalen LISP-Variabeln `*paths*` eine Continuation mit `(choose (rest choices))` hinzugefügt und dann das erste Element von `choices` zurückgegeben.

Wird `fail` aufgerufen, so ruft die Funktion die erste Continuation aus `*paths*` auf und entfernt diese aus `*paths*`. `*paths*` ist ein Stack, dadurch wird bei einem Aufruf von `fail` immer zur letzten möglichen Alternative in der Suche zurückgelaufen, d.h. ein Backtracking erfolgt.

Wenn es keine Alternative mehr gibt, wird `no-solution` ausgegeben. Durch die Formulierung als Continuation wird dies nicht dem letzten `choose`, das `fail` aufrief, zurückgegeben, sondern wird direkt auf dem `top-level` ausgegeben.

[7] Siehe Abschnitt 3.3.
[8] Siehe Abschnitt 2.2.

```
(define *paths* ())
(define failsymbol 'no-solution)

(define (choose choices)
  (if (null? choices)
      (fail)
    (call-with-current-continuation
      (lambda (cc)
        (set! *paths*
              (cons (lambda ()
                      (cc (choose (cdr choices))))
                    *paths*))
        (car choices)))))

(define fail ())

(call-with-current-continuation
  (lambda (cc)
   (set! fail
         (lambda ()
           (if (null? *paths*)
               (cc failsymbol)
             (let ((p1 (car *paths*)))
               (set! *paths* (cdr *paths*))
               (p1)))))))
```

Abbildung A.3.: Implementierung von Nondeterminismus in SCHEME

B. Klassischer Kontrapunkt: ein ausführlicheres Beispiel

Das folgende Beispiel stellt eine ausführlichere Fassung dessen dar, was der Abschnitt 4.4 vorstellt. Das Programm realisiert einen einfachen zweistimmigen Kontrapunkt mit verschiedenen Constraints zur Stimmführung, zum Rhythmus und zum Zusammenklang. Dies Beispiel demonstriert die prinzipiellen Möglichkeiten von Arno. Es ist unvollständig: die Regeln zur Dissonanzbehandlung könnten verfeinert, es könnten Constraints zum Schließen mit einer Kadenz usw. formuliert werden. Ein mögliches Ergebnis zeigt Abb. 4.16.

```
(in-package :cm)

#|
(thread stimme ())
(one-value (stimme #!stimme)

(merge zweiStimmig ())
(one-value (zweiStimmig #!zweiStimmig))

(thread stimm-segment-bj ())
(one-value (stimm-segment-bj #!stimm-segment-bj :min-ton 60 :max-ton 72 :channel 1))

(merge zweiStimmig-bj ())
(one-value (zweiStimmig-bj #!zweiStimmig-bj))
|#

;;;
;;; drei verschiedene Fassungen: einfach, mit Pausen zwischen Abschnitten, mit BJ
;;;

(defcontain stimme
    :further-args (&key min-ton max-ton channel)
    :content-type (object midi-note rhythm 0)
    :let* ((number (between 17 27)))
    :number number
    :content-setting
    (progn
      (set-object (current-object)
                  'rhythm (a-shuffled-expr (a-member-of '(16 8 6 4 3 2 1)))
                  ;; unary constraint c-major, mit geg. Stimmumfang
                  'note (note (mode-degree
                               (a-shuffled-expr
                                (an-integer-between min-ton max-ton)) c-major))
                  'cm::channel channel))
    :avoid '(nicht-erlaubter-zsklang? parallelen? not-in-rhythm-env? no-smooth-rhythm?
             punktierte-rhythmen-falsch? taktpos-falsch? zu-wenig-schritte?
```

```
            nicht-erlaubte-Intervalle? not-ballistic? no-gegenbew-nach-gr-sprung?
            dreiklangs-brechung? dupl-peaks? dissonanz-zw-wendepunkten?)
    )

(defcontain zweiStimmig
    :content-type  (make-object 'thread)
    :number 2
    :content-setting
    (case i-with
      (0 (stimme  (current-object) :min-ton 48 :max-ton 65 :channel 0))
      (1 (stimme  (current-object) :min-ton 50 :max-ton 67 :channel 1)))
    )

(defcontain stimm-segment
    :beginning (list (object rest
                      rhythm (expt 2 (a-shuffled-expr (an-integer-between 1 4)))))
    :further-args (&key min-ton max-ton channel)
    :content-type (object midi-note rhythm 0)
    :let* ((number (between 5 13)))
    :number number
    :content-setting
    (progn
      (set-object (current-object)
                  'rhythm (a-shuffled-expr (a-member-of '(16 8 6 4 3 2 1)))
                  'note (note (mode-degree
                               (a-shuffled-expr
                                (an-integer-between min-ton max-ton)) c-major))
                  'cm::channel channel))
    :avoid '(nicht-erlaubter-zsklang? parallelen? not-in-rhythm-env? no-smooth-rhythm?
            punktierte-rhythmen-falsch? taktpos-falsch? zu-wenig-schritte?
            nicht-erlaubte-Intervalle? not-ballistic? no-gegenbew-nach-gr-sprung?
            dreiklangs-brechung? dupl-peaks? dissonanz-zw-wendepunkten?)
    )

(defcontain stimme-segmented
    :further-args (&key min-ton max-ton channel)
    :content-type (make-object 'thread)
    :number 3
    :content-setting (stimm-segment (current-object)
                                    :min-ton min-ton :max-ton max-ton :channel channel)
    )

(defcontain zweiStimmig-segmented
    :content-type  (make-object 'thread)
    :number 2
    :content-setting
    (case i-with
      (0 (stimme-segmented  (current-object) :min-ton 48 :max-ton 65 :channel 0))
      (1 (stimme-segmented  (current-object) :min-ton 50 :max-ton 67 :channel 1)))
    )


(defcontain stimm-bj
    :bj? T
    :further-args (&key min-ton max-ton channel)
    :content-type (object midi-note rhythm 0)
    :let* ((number (between 7 17)))
    :number number
```

```
    :content-setting
    (progn
      (set-object (current-object)
                  'rhythm (a-shuffled-expr (a-member-of-bj '(16 8 6 4 3 2 1)))
                  'note (note (mode-degree
                               (a-shuffled-expr
                                (an-integer-between-bj min-ton max-ton)) c-major))
                  'cm::channel channel))
    :avoid '(nicht-erlaubter-zsklang? parallelen? not-in-rhythm-env? no-smooth-rhythm?
             punktierte-rhythmen-falsch? taktpos-falsch? zu-wenig-schritte?
             nicht-erlaubte-Intervalle? not-ballistic? no-gegenbew-nach-gr-sprung?
             dreiklangs-brechung? dupl-peaks? dissonanz-zw-wendepunkten?)
    )


(defcontain zweiStimmig-bj
    :bj? T
    :content-type  (make-object 'thread)
    :number 2
    :content-setting
    (case i-with
      (0 (stimme-bj (current-object) :min-ton 48 :max-ton 65 :channel 0))
      (1 (stimme-bj (current-object) :min-ton 50 :max-ton 67 :channel 1)))
    )

(defvar c-major)
(setf c-major (transpose (mode major 2 2 1 2 2 2 1) 'c))

;; wenn zusätzliche Leittöne eingesetzt werden sollen, gegen c-major auszutauschen
;; und Constraint leitton-falsch? einzufügen
(defvar c-major-extended)
(setf c-major-extended (transpose (mode major-extended 1 1 2 1 1 1 1 1 2 1) 'c))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; Constraints
;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;
;;; zur Stimmfuehrung
;;;

(defmethod leitton-falsch? ((note midi-note))
  ;; steht Zielton vor und nach Leitton?
  ;; Leitton gehoert spaeter in Kadenz und unerhoehter glnamiger davor zu vermeiden
  (let*-when ((prev (previous-note note))
              (degree-note (degree (get-note note)))
              (degree-prev (degree (get-note prev)))
              (interval (- degree-note  degree-prev)))
      ;; wenn note Leitton muss interval kl. 2 runter
      ;; wenn prev Leitton muss interval kl 2 hoch
      (unless (cond ((member (mod degree-note 12) '(1 6 8)) (= interval -1))
                    ((member (mod degree-prev 12) '(1 6 8)) (= interval 1))
                    (T T))
        t)))
;;;        prev)))                                ;BJ zu prev quatsch?
```

```
(defmethod leitton-falsch? ((rest rest))
  NIL)


(defmethod nicht-erlaubte-Intervalle? ((note midi-note))
  ;; keine Wertung: welche Intervale wie oft
  ;; keine Tonwiederholung erlaubt
  (let*-when ((prev (previous-note note))
              (intervall (- (degree (get-note note))
                            (degree (get-note prev)) )))
      (unless (or (< 0 (abs intervall) 5)
                  (member (abs intervall) '(7 12))
                  (= intervall 8))
        t)))
;;;        prev)))

(defmethod nicht-erlaubte-Intervalle? ((rest rest))
  NIL)

(defmethod no-gegenbew-nach-gr-sprung? ((note midi-note))
  ;; macht "ballistische Kurven" unmoeglich?
  (let*-when ((prev (previous-note note))
              (prev-prev (previous-note prev))
              (int-1 (- (degree (get-note prev)) (degree (get-note prev-prev)) ))
              (int-2 (- (degree (get-note note)) (degree (get-note prev)) )))
      ;; wenn erstes Intervall Sex oder Oktav, muss naechstes Intervall entgegen sein
      ;; max. kl. Terz nach aufwaerts Sprung,
      ;; max. gl. Intervall wieder hoch nach Abwaertssprung
      (unless
          (cond ((> int-1 7) (<= -3 int-2 0))
                ((< int-1 -7) (<= 0 int-2 int-1))
                (T T))
        t)))
;;;        prev)))

(defmethod no-gegenbew-nach-gr-sprung? ((rest rest))
  NIL)

(defmethod not-ballistic? ((note midi-note))
  (let*-when ((prev (previous-note note))
              (prev-prev (previous-note prev))
              (int-1 (- (degree (get-note prev)) (degree (get-note prev-prev)) ))
              (int-2 (- (degree (get-note note)) (degree (get-note prev)) )))
      ;; nach Sprung/Schritt hoch ist folgendes Intervall kleiner, Richtung egal
      ;; vor Sprung hoch ist voriges Intervall abwaerts oder groesser
      ;; nach Sprung/Schritt runter ist folgendes Intervall abwaerts "gr" od. aufwaerts
      ;; vor Sprung runter ist voriges Intervall kleiner,  Richtung egal
      ;; (damit keine Rueckkehr zum selben Ton moeglich!)
      ;; (pos. Intervall hoch, neg. runter)
      (unless
          (cond ((> int-1 3) (> (abs int-1) (abs int-2)))
                ((> int-2 3) (or (< int-1 0)
                                 (> int-1 int-2)))
                ((< int-1 -3) (or (< int-2 int-1)
                                  (> int-2 0)))
                ((< int-2 -3) (< (abs int-1) (abs int-2)))
                (T T))
```

```
        t)))
;;;        prev)))

(defmethod not-ballistic? ((rest rest))
  NIL)

(defmethod dreiklangs-brechung? ((note midi-note))

  (let*-when ((prev (previous-note note))
              (prev-prev (previous-note prev))
              (int-1 (- (degree (get-note prev)) (degree (get-note prev-prev)) ))
              (int-2 (- (degree (get-note note)) (degree (get-note prev)) )))
      (when (member (list (abs int-1) (abs int-2))
                    '((3 4) (4 3) (3 3) (4 4)) :test #'equal)
        t)))
;;;        prev)))

(defmethod dreiklangs-brechung? ((rest rest))
  NIL)

(defmethod zu-wenig-schritte? ((note midi-note))
  ;; avoid: gibt nur T oder NIL zurueck...
  ;; sind von anzahl Toenen min. Verhaeltnis Schritte?
  ;; ist ziemlich ineffizient --- aber funktioniert.
  (let* ((anzahl 4)
         (verhaeltnis 2/3)
         (last-ints (last-n-intervals note anzahl)))
    (and (>= (length last-ints) anzahl)
         (<= (nr-schritte-of-ints last-ints)
             (* anzahl verhaeltnis)))))

(defmethod zu-wenig-schritte? ((rest rest))
  NIL)

(defun nr-schritte-of-ints (ints)
  ;; aux
  ;; zaehlt das Vorkommen von -2, -1, 1, und 2 in ints
  (+ (count 2 ints)
     (count 1 ints)
     (count -1 ints)
     (count -2 ints)))



(defmethod dissonanz-zw-wendepunkten? ((note midi-note))
  (let*-when ((prev-turn (letzter-richtungswechsel note))
              (prev-prev-turn (letzter-richtungswechsel prev-turn))
              (turn-interval (- (degree (get-note prev-turn))
                                (degree (get-note prev-prev-turn)))))
      (when (member (abs turn-interval) '(6 10 11 13 14))
        prev-turn)))

(defmethod dissonanz-zw-wendepunkten? ((rest rest))
  NIL)

(defmethod dupl-peaks? ((note midi-note))
  (let*-when ((prev-turn (letzter-richtungswechsel note))
              (prev-prev-turn (letzter-richtungswechsel prev-turn))
```

```
                (prev-prev-prev-turn (letzter-richtungswechsel prev-prev-turn)))
270       ;; die Tonhoehen zweier Wendepunkte in die gl. Richtung
          ;; sollen sich nicht gleichen
          ;; (deshalb einer ausgelassen)
          (when (= (degree (get-note prev-turn))
                   (degree (get-note prev-prev-prev-turn)))
            prev-turn)))

    (defmethod dupl-peaks? ((rest rest))
      NIL)

280 ;;;
    ;;; zum Rhythmus
    ;;;

    (defmethod no-smooth-rhythm? ((note midi-note))
      ;; naechste note mindestens halb und hoechsten doppelt so lang wie Vorgaenger
      (let*-when ((prev (previous-note note)))
          (unless  (<= 1/3 (/ (get-rhythm prev) (get-rhythm note)) 3)
            prev)))

290 (defmethod no-smooth-rhythm? ((rest rest))
      NIL)

    (defmethod punktierte-rhythmen-falsch? ((note midi-note))
      ;; punktierter Wert kann nur auf volle oder halbe Zeit kommen
      ;; ihm folgt der ihn vervollstaendigende Wert sofoert
      (let*-when ((prev (previous-note note))
                  (prev-rhythm (get-rhythm prev))
                  (note-rhythm (get-rhythm note)))
          ;; wenn prev punktiert: ist note genau 1/3 davon?
300       ;; wenn note punktiert: darf min. um Haelfe von ihrem Kombi
          ;; mit folgender  im Takt verschoben stehen
          ;; (Fall von pkt. Halben mit 2 Vierteln damit nicht erlaubt)
          (unless
            (cond  ((integerp (/ prev-rhythm 3))  (= (* note-rhythm 3) prev-rhythm))
                   ((integerp (/ note-rhythm 3)) (integerp (/ (slot-value note 'time)
                                                              (* note-rhythm 2/3))))
                   (T T))
            prev)))

310 (defmethod punktierte-rhythmen-falsch?  ((rest rest))
      NIL)


    (defmethod taktpos-falsch? ((note midi-note))
      ;; avoid : gibt nur T oder NIL zurueck
      ;; note darf um min halbe Laenge im Takt verschoben stehen
      ;; pkt. Noten extra getestet
      (let*-when ((note-rhythm (get-rhythm note)))
          (if (integerp (/ note-rhythm 3)) NIL
320         (not (integerp (/ (slot-value note 'time)
                              (/ note-rhythm 2) ))))))

    (defmethod taktpos-falsch? ((rest rest))
      NIL)
```

```
(defmethod not-in-rhythm-env? ((note midi-note))
  ;; um die globale Dauernentw. kontollieren zu koennen
  ;; leider abhaengig von Anzahl der Toene in Thread,
  ;; da Auflösung der Env abhängig von Anzahl verschieden genau
  (let* ((rhyhtm-env '((0 8 16)
                       (.2 2 8)
                       (.5 1 6)
                       (.7 1 4)
                       (.8 1 6)
                       (.9 2 8)
                       (1 8 16)))
         (max-env (upper-env rhyhtm-env))
         (min-env (lower-env rhyhtm-env))
         (note-anzahl (object-count (cm::the-container note)))
         (note-nr (object-position note))
         (max-env-wert (env-value note-nr note-anzahl max-env))
         (min-env-wert (env-value note-nr note-anzahl min-env)))
    (not (<= min-env-wert (get-rhythm note) max-env-wert))))

(defmethod not-in-rhythm-env? ((rest rest))
  NIL)


;;;;; wenn mehr als n Toene einstimmig, springe zurueck zur letzten Pause
;;;(defmethod zu-lange-einstimmig? ((note midi-note))


;;;
;;; zu Zusammenklang
;;;


;; die verschiedenen Dissonanzbehandlungen ließen sich hier
;; als einzelne Tests mit cond am Ende einbinden
(defmethod nicht-erlaubter-zsklang?  ((note midi-note))
  ;; nur fuer zweistimmigkeit (wegen der erlaubten Intervalle)
  ;; moeglich: ausgehend von tiefster Note,
  ;; (aber was, wenn diese erst spaeter instaziiert wird -> Stimmkreuzung...)
  ;; simultan-notes gibt nur erste Dissonanz zurueck!!
  ;; kein Einklang erlaubt
  (let*-when ((note-degree (degree (get-note note)))
              (simultans
               (remove-if-not
                #'(lambda (simultan)
                    ;; ist schon Notenwert gesetzt?
                    (cm::careful-slot-value simultan 'note))
                (simultan-notes note)))
              (simultan-dissonances
               (mapcar
                #'(lambda (simultan)
                    (let ((interval (- (degree (get-note simultan)) note-degree)))
                      ;; wenn keine Konsonanz, gib simultan zurueck, sonst NIL
                      (unless (member (abs interval) '(3 4 7 8 9 12 15 16))
                        simultan)))
                simultans)))
    ;; wenn irgendeiner nicht NIL, gib den ersten zurueck
    (when (or simultan-dissonances)
      (first (remove-if #'null simultan-dissonances)))))
```

```
(defmethod nicht-erlaubter-zsklang?  ((rest rest))
  NIL)

(defmethod parallelen? ((note midi-note))
  ;; nur fuer Zweistimmigkeit,
  ;; da nur von einer note-beginning-same-time ausgegangen wird
  ;; aus gleicher Richtung in Prime/Quinte/Oktave (verdeckte und offene Quinte) ?
  ;; sollen verdeckte lieber erlaubt sein?
  (let*-when ((sim (note-beginning-same-time note))
              (prev-note (previous-note note))
              (prev-sim (previous-note sim))
              (note-degree (degree (get-note note)))
              (sim-degree (degree (get-note sim)))
              (vertical-interval (abs (- sim-degree note-degree)))
              (interval1 (- (degree (get-note prev-note)) note-degree))
              (interval2 (- (degree (get-note prev-sim)) sim-degree)))
      ;; unerlaubte Parallelen,
      ;; wenn member bewusstes intervall und Vorintervalle aus gl. Richtung
      (when (and (member vertical-interval '(0 7 12))
                 (or (and (> interval1 0) (> interval2 0))
                     (and (< interval1 0) (< interval2 0))))
        sim)))                                ; BJ Ziel (nach note)

(defmethod parallelen? ((rest rest))
  NIL)

;;;(defmethod akzent-parallele? ((note midi-note))
;;;
;;;  ;; wenn Note Vorhaltnote hat (d.h. auch, Note ist auf schwerer Zeit),
;;;  ;; und beide vorigen Intervalle waren in gl. Richtung
;;;  ;; keine Oktav/Prim zu dieser
;;;  (let ((vorhalt (vorhalt-note note)))
;;;    (when vorhalt
;;;      ))
;;;  )
;; auf schwere Zeit in Oktave (Akzentparallele) ?

;;;
;;; utils
;;;

(defmethod letzter-richtungswechsel ((note midi-note))
  ;; wo letzter Vorzeichenwechsel der Folgeintervalle in Stimme vor note?
  (let*-when ((prev (previous-note note))
              (prev-prev (previous-note prev))
              (int-1 (- (degree (get-note prev)) (degree (get-note prev-prev)) ))
              (int-2 (- (degree (get-note note)) (degree (get-note prev)) )))
      (if (or (and (< int-1 0) (> int-2 0))
              (and (> int-1 0) (< int-2 0)))
          prev
        (letzter-richtungswechsel prev))))

(defmethod last-n-intervals ((note midi-note) (n number) &optional intervals)
  (let ((prev (previous-note note)))
    (if (or (not prev) (zerop n))
        (reverse intervals)
      (let ((interval (- (degree (get-note note))
```

```
                          (degree (get-note prev)) )))
        (last-n-intervals prev (1- n) (cons interval intervals))))))

;; simultan-object-same-time

(defmethod objects-beginning-same-time
    ((event rhythmic-element) &key (only-type t) container)
  (remove-if-not #'(lambda (sim) (same-time? sim event))
                 (simultan-objects event :only-type only-type :container container)))

(defmethod same-time? ((event1 rhythmic-element) (event2 rhythmic-element))
  ;; returns list of start and end time points of timed-event
  ;; -> error, if time slot is unset
  ;; if no duration is set, the rhythm value is taken instead
  (let* ((time1 (cm::careful-slot-value event1 'cm::time))
         (time2 (cm::careful-slot-value event2 'cm::time)))
    (if (and time1 time2)
        (= time1 time2)
      (error "time-slot of ~a or ~a unset, exec (run-object container) first"
             event1 event2))))

(defmethod vorhalt? ((note rhythmic-element) (vorgehalten rhythmic-element))
  ;; note auf schwerer Zeit
  ;; vorgehalten in Beginn von note uebergebunden und waerend dauer note zuende
  (let* ((time-note (cm::careful-slot-value note 'cm::time))
         (time-vorg (cm::careful-slot-value vorgehalten 'cm::time)))
    (unless (and time-note time-vorg)
      (error "time-slot of ~a or ~a unset, exec (run-object container) first"
             note vorgehalten))
    (let ((end-note (+ (get-rhythm note) time-note))
          (end-vorg (+ (get-rhythm vorgehalten) time-vorg)))
      (and (integerp (/ time-note 2))        ; schwere Zeit
           (<  time-vorg time-note  end-vorg end-note)) )))

(defmethod note-beginning-same-time ((note rhythmic-element))
  (first (objects-beginning-same-time note :only-type 'note)))

(defmethod vorhalt-note ((note rhythmic-element))
  (first (remove-if-not #'(lambda (sim) (vorhalt? note sim))
                        (simultan-objects note :only-type 'note))))
```

C. Lispcode von Arno

Im folgenden wird der vollständige Quelltext von ARNO mitgeteilt. Die Entwicklung von ARNO ist mit der Abgabe der Diplomarbeit nicht beendet, dem Code ist dieses Work-in-Progress durchaus anzusehen. Manches würde der Autor inzwischen anders lösen, einiges wurde für einen sehr speziellen Zweck entwickelt, vieles kann noch ergänzt oder verbessert werden. Konkrete Ideen zur Effizienzsteigerung wie auch zu Erweiterungen enthält das Kapitel 6.

C.1. Die Datei *\arno-kernel.lisp:

```
;       $Id: arno-kernel.lisp,v 1.27 1999/12/29 12:24:08 t Exp $

;;*****************************************************************************
;;
;;   Copyright (c) 1999 by Torsten Anders. All rights reserved.
;;   For suggestions, comments and bug reports email to: torsten.anders@hfm.uni-weimar.de
;;
;;   Copyright (c) of Common Music 89-97, 98 Heinrich Taube. All rights reserved.
;;   Copyright (c) of Screamer 1991 Massachusetts Institute of Technology. All rights reserved.
;;                             1992, 1993 University of Pennsylvania. All rights reserved.
;;                             1993 University of Toronto. All rights reserved.
;;
;;   This program is free software; you can redistribute it and/or modify
;;   it under the terms of the GNU General Public License as published by
;;   the Free Software Foundation; either version 2 of the License, or
;;   (at your option) any later version.
;;
;;*****************************************************************************

(in-package :arno)

(defvar *arno-kernel*                          ; folgende Symbole werden exportiert
    '(container highest-container i-with
      defcsp defcontain))                ; eine Funktion print-statistics ist bereits in CM definiert!

;; waere eine reader- und writer-function fuer i-with, i-without und container sinnvoll (als macro)?

;; ! Problem: beginning wird evaluiert, bevor es in Hierachie eingebunden wird,
;; d.h. es kann beim Füllen eines Containers, definiert in beginning,
;; nicht richtig auf z.B. simultan-objects referenziert werden!

;; alle Slots werden z.Z. global beschrieben, ein variated?-slot koennte local sein!
;; brauchts concluding-expr?

;; dass fulfil und avoid gequoted werden muessen kann geaendert werden:
;; in Test, ob Constraints übergeben werden Ausdruck (quote fulfil)
```

```
(defmacro defcontain
    (funname
     &key content-type content-setting beginning number fulfil avoid bj?
          let* further-args beginning-in-loop? concluding-expr)
  ;; to minimize the resulting function, as much as possible is expressed in backquotes before
  ;; to be omited if not necessary.
  ;; Variables bound in the body of defun are called in backquoted expr defined before...
  (let (;; will be inserted in let* of defun if beginning is not nil
        (beg-obj-let*
         (when beginning
           `((beginning-objects ,beginning)
             (beginning-length (length beginning-objects)))))
        ;; returns the empty content for the container-initialisation
        (unset-content-expr
         (cond ((and number beginning)
                `(append beginning-objects
                         (make-unset-objects ,content-type ,number)))
               (number `(make-unset-objects ,content-type ,number))
               (beginning 'beginning-objects)
               (t (error
                   "whether number nor beginning given in defcontain ~s"
                   funname))))
        ;; returns loop number for dotimes in defun
        (loop-number-expr
         (cond ((and number beginning beginning-in-loop?)
                `(+ ,number beginning-length))
               ((and beginning beginning-in-loop?)  'beginning-length)
               (number number)
               (t (progn
                    (warn
                     "number or both beginning and beginning-in-loop? not given in defcontain ~s"
                     funname)
                    0))))
        ;; loop counter i in two variables for with or without beginning (i-with, i-without)
        (i-with-expr (if beginning
                         `(+ i-without beginning-length)
                       'i-without))
        ;; two test-expr which can cause backtracking
        ;; two variations handle fulfil (must) and avoid (must not).
        (fulfil-test-expr
         (when fulfil
           (progn
             (when bj? (warn "fulfil predicates can not backjump in defcontain ~S" funname))
             `((let ((current-object (nth-object i-with container)))
                 (unless (all-true? current-object ,fulfil)
                                    ;(progn (unset-object current-object)
                           (fail)))))))
        ;; backjumping works only for to avoid predicates:
        ;; the to-avoids return the backjumping target if test fails, i.e. returns something
        (avoid-test-expr
         (cond ((and bj? avoid)
                `((let ((current-object (nth-object i-with container)))
                    (unless (non-true? current-object ,avoid)
                      (progn
                        (set-pointer *bj-pointer*
                                     (bj-target current-object ,avoid))
                        (unset-object current-object)
```

```
                        (fail)))
                    (set-pointer *bj-pointer* NIL))))
                (avoid
                 `((let ((current-object (nth-object i-with container)))
                    (unless (non-true? current-object ,avoid)
                                    ;(progn (unset-object current-object)
                            (fail))))))))
      (if concluding-expr (setf concluding-expr `(,concluding-expr)))
      ;; container will be the container to be filled
      ;; further-args is key arg of defcontain
      `(defun ,funname (container ,@further-args)
;;;     (format t "~%container ~a, dessen container ~a" container (cm::the-container container))
        (when *write-statistics* (if (cm::the-container container) ; if not, container is in top-level
                                     (append-statistics (append ,fulfil ,avoid))
                                   (init-statistics (append ,fulfil ,avoid))))
        (let* (,@let*                       ; let* is key arg of defcontain
               ,@beg-obj-let*
               (highest-container
                (first (list-object-containers container)))
              unset-content i-with loop-number )
          (screamer:local               ; for undoable side effects (!)
           (setf unset-content ,unset-content-expr)
           (set-container-objects container unset-content) ; will always be global, not undoable
           (setf loop-number ,loop-number-expr)
           ;; counters i-with and i-without local, incr could be unwind by backtracking
           (dotimes (i-without loop-number)
;;;          (break "i-without: ~S in fun ~S" i-without (quote ,funname))
;;;          (format t "~%in fun ~a, container ~a" (quote ,funname) container)
             (setf i-with ,i-with-expr)
             (run-object highest-container) ; to update the time slots of all events
             ;; content-setting contains expr which really does the job:
             ;; calls set-object on current note or inserts an inner container.
             ;; current object in loop should be accessed with (nth-object i-with container)
             ;; content-setting is key arg of defcontain
             ,content-setting
             ;; after setting, content will be tested
             ;; if tests not succeed, backtracking will happen.
             ,@fulfil-test-expr
             ,@avoid-test-expr
             ;; sometimes some cleaning should be done...
             ,@concluding-expr))
           container))))


;; ganz allg., keine schleife, nicht auf Container beschraenkt,
;; erzeugt keine Objecte automatisch
;; fehlt fulfil-arg
(defmacro defcsp
    (funname &key args let* content-setting fulfil avoid)
  `(defun ,funname (,@args)
     (when *write-statistics* (append-statistics ,avoid))
     (let* (result ,@let*)
       (setf result ,content-setting)
       (when (true? ,avoid)
         (unless
             (non-true? result ,avoid)
           (fail)))
       (when (true? ,fulfil)
```

```
          (unless
              (all-true? result ,fulfil)
            (fail)))
        result)))

(defmacro defcsp-bj
    (funname &key args let* content-setting avoid bj-targets)
  `(defun ,funname (,@args)
     (let* (result ,@let*)
       (setf result ,content-setting)
       (when (true? ,avoid)
         (unless
             (non-true? result ,avoid)
           (progn
             (set-pointer *bj-pointer* ,bj-targets)
             (fail)))
         (set-pointer *bj-pointer* NIL))
       result)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defvar *bj-pointer* (make-object 'bj-pointer))


;; sortiere die Ausgaben der Praedikate, die Fehlschlagursachen:
;; innehalb einer Ausgabeliste abfallend, d.h. after-...,
;; sortiere NILs aus
;; dann von jedem den ersten aufsteigend
;; wenn schon mal ein dichteres Ziel war, nimm das
;; Predicates werden alle nochmal getestet: nicht effizient
(defmethod bj-target ((curr-obj cm::timed-object) predicates)
  (let (result)
    (dolist (predicate predicates)
      (setf result
        (cons                                ; durch flat Liste garantiert
         (first (sort (remove NIL
                              (flat (funcall predicate curr-obj))) #'after-in-seeking?))
         result)))
    (list curr-obj
          (first
           (sort
            (remove NIL
                    (cons (first (sort (remove NIL result) #'before-in-seeking?))
                          (remove curr-obj (pointer *bj-pointer*))))
            #'after-in-seeking?)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; erstmal ganz einfach: gehe von einem defcontain aus.
;; Spaeter zaehlen alle defcontains in den selben Zaehler

;; is an assoc list (object hat nicht beliebige Anzahl von slots)
(defvar *defcontain-statistics* NIL)
;; flag - soll Statistik aufgemacht werden?
(defvar *write-statistics* NIL)

;; erstmal ganz primitiv...
;;;(defun print-statistics ()
```

```
;;;  (pprint *defcontain-statistics*))
(defun print-statistics ()
  (let ((statistics-copy (copy-tree *defcontain-statistics*))
        (count-statistics '(()))
        (sum-statistics '(())))
    (mapcar #'(lambda (pair)
                (let* ((assoc-key (first pair))
                       (key-val-in-sum (assoc assoc-key sum-statistics)))
                  (if key-val-in-sum
                      (progn
                        (rplacd (assoc assoc-key sum-statistics)
                                (+ (rest key-val-in-sum) (rest pair)))

                        (rplacd (assoc assoc-key count-statistics)
                                (1+ (rest (assoc assoc-key count-statistics)))))
                    (setf sum-statistics (cons pair  sum-statistics)
                          count-statistics (cons (cons (first pair) 1) count-statistics)))))
            statistics-copy)
    (format t "~%happened fails in defcontain/defcsp:")
    (mapcar #'(lambda (sum-pair)
                (format t "~%~a times fail happened in ~a  occurences of ~a."
                        (rest sum-pair)
                        (rest (assoc (first sum-pair) count-statistics))
                        (first sum-pair)))
            (butlast sum-statistics))))

(defun toggle-statistics (predicate)
  (setq *write-statistics* predicate)
  (format t "~%defcontain statistics are now ~a." (if predicate "enabled" "disabled")))

(defun init-statistics (predicates)
  (setf *defcontain-statistics*
    (mapcar #'(lambda (in) (cons in 0)) predicates)))

(defun append-statistics (predicates)
  ;; neue Praedikate werden vorn angehaengt, assoc aendert usw. immer diese,
  ;; aeltere Werte sind history
  (setf *defcontain-statistics*
    (append (mapcar #'(lambda (in) (cons in 0)) predicates)
            *defcontain-statistics*)))

(defun incr-statistic-counter (predicate)
  (rplacd (assoc predicate *defcontain-statistics*)
          (1+ (rest (assoc predicate *defcontain-statistics*)))))

(defun update-defcontain-statistics (boolean predicates)
  (mapcar #'(lambda (bool pred)
              (unless bool (incr-statistic-counter pred)))
          boolean
          predicates))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; noch nachbaun: Moeglichkeit, jeder Klausel einen Commentstring zu uebergeben (debuging)

;;;(defmacro defconstraint (constrain-name obj-and-class &key refs conds)
;;;  ;; references gibt (wie in let*) jeder ref einen Namen, (d.h. Bezuege auf vorige moegl.)
```

```
;;;  ;; testet nacheinander, ob references gebunden werden koennen, wenn nicht: "no such reference" -> T
;;;  ;; wenn ja, dann conditions mit entsprechender aktion (ein Paedikat)
;;;  ;; triff keine condition zu: "no condition met" -> T
;;;
;;;  ;; Erweiterung sollte einfaches Umschalten zu bj ermoeglichen...
;;;    `(defmethod ,constrain-name ((,obj ,class))
;;;       (when-binds*
;;;        ',refs ',conds ))))
```

C.2. Die Datei *\object-utils.lisp

```
;        $Id: object-utils.lisp,v 1.14 1999/12/29 12:25:14 t Exp $

;;***************************************************************************
;;
;;   Copyright (c) 1999 by Torsten Anders. All rights reserved.
;;   For suggestions, comments and bug reports email to: torsten.anders@hfm.uni-weimar.de
;;
;;   Copyright (c) of Common Music 89-97, 98 Heinrich Taube. All rights reserved.
;;   Copyright (c) of Screamer 1991 Massachusetts Institute of Technology. All rights reserved.
;;                             1992, 1993 University of Pennsylvania. All rights reserved.
;;                             1993 University of Toronto. All rights reserved.
;;
;;   This program is free software; you can redistribute it and/or modify
;;   it under the terms of the GNU General Public License as published by
;;   the Free Software Foundation; either version 2 of the License, or
;;   (at your option) any later version.
;;
;;***************************************************************************

(in-package :arno)

(defvar *object-utils*                        ; folgende Symbole werden exportiert
    '(list-object-containers list-container-objects set-container-objects
      get-note set-note get-rhythm set-rhythm
      simultan-objects simultan-notes simultan-chord
      previous-objects previous-object previous-note all-previous-objects later-objects
      first-object? second-object? last-object?
      chronical-sort-objects thread-dur time-after-previous-container
      current-object current-object-including-beginning
      objects-without-container kill-unused))


(defmethod list-object-containers ((input element))
  ;; lists all containers and super containers of object
  ;; if object belongs to more containers in one level list-object-containers returns all of them
  (let ((super-container (object-containers input)))
    (flat
     (mapcar #'list-object-containers super-container))))

(defmethod list-object-containers ((input container))
  (let ((super-container (object-containers input)))
    (append
     (flat
      (mapcar #'list-object-containers super-container))
```

```
      (list input))))

;; lists all and only the events of container (also in its subcontainers)
(defmethod list-container-objects ((input element))
  input)
(defmethod list-container-objects ((input container))
  (let ((container-content (container-objects input)))
    (flat
     (mapcar #'list-container-objects container-content))))

(defmethod set-container-objects ((container container) objects)
  ;; container will be emptied first, then the objects will be inserted
  (cm::remove-and-unlink-all-objects container)
  (add-objects objects container))

(defmethod replace-object
    ((old-object cm::timed-object) (new-object cm::timed-object))
  ;; replaces old-object in container with new-object
  (let ((container (cm::the-container old-object))
        (position (object-position old-object)))
    (remove-object position container)
    (add-object new-object container position)))

(defmacro make-unset-objects (typ-def number)
  ;; typ-def beschr. object-def fuer Initialisierung, z.B. (object note))
  `(let (result)
     (dotimes (i ,number (reverse result))
       (push ,typ-def result))))

;;-----------------------------------------------------------------------------


(defmethod get-note ((note note))
  (cm::careful-slot-value note 'note))

(defmethod set-note ((note note) val)
  (setf (slot-value note 'note) val))

(defmethod get-rhythm ((note rhythmic-element))
  (cm::careful-slot-value note 'rhythm))

(defmethod set-rhythm ((note rhythmic-element) val)
  (setf (slot-value note 'rhythm) val))

(defmethod unset-object ((obj rhythmic-element))
  ;; unsets all slots of object except time, flags, container
  ;; (for run-object the rhythm slot must be initialized)
  ;; was ist mit ins (csound-note), rhythm so ok?
  (mapcar
   #'(lambda (slot)
       (slot-makunbound-using-class (class-of obj) obj slot))
   (remove-if #'(lambda (slot)
                  ;; sind slots Symbole oder Objects, waer bei Symbolen remove einfacher?
                  (member slot '(cm::TIME cm::FLAGS cm::CONTAINER)))
              (CLOS::OBJECT-INSTANCE-SLOT-NAMES obj)))
  (set-rhythm obj 0))

;;-----------------------------------------------------------------------
```

```
;; utils for time domain

(defmethod simultan-objects ((timed-event rhythmic-element)
                             &key (only-type t) container)
  ;; returns list of objects which sound in the same time span as given event.
  ;; with given only-type key arg only the objects of the desired object typ will be returned.
  ;; key arg container to search all simultan obj to given event in that container.
  ;; default is highest container of given event (if there are more the first is taken).
  ;; all events in container have to get a time-stamp befor (run-object).
  (when (not container)
    (setf container (first (list-object-containers timed-event))))
  (let ((time-interval (object-time-interval timed-event))
        ;; event-liste without input timed-event
        (object-list
         (remove timed-event
                 (filter-only-type
                  (list-container-objects container) only-type)))
        result)
    (dolist (curr-event object-list)
      (let ((curr-interval (object-time-interval curr-event)))
        (when (still-in-time-interval? time-interval curr-interval)
          (setf result (cons curr-event result)))))
    (reverse result)))

;; aux for simultan-objects
(defmethod object-time-interval ((timed-event rhythmic-element))
  ;; returns list of start and end time points of timed-event
  ;; -> error, if time slot is unset
  ;; if no duration is set, the rhythm value is taken instead
  (let* ((time (cm::careful-slot-value timed-event 'cm::time))
         (dur-slot (cm::careful-slot-value timed-event 'cm::duration))
         (dur (if dur-slot dur-slot
                  (cm::careful-slot-value timed-event 'rhythm))))
    (if time
        (list time (+ time dur))
      (error "time-slot of ~a unset, exec (run-object container) first" timed-event))))

;; aux for simultan-objects
(defun still-in-time-interval? (interval1 interval2)
  ;; is beginning of time interval2 in interval1
  ;; or is beginning of time interval1 in interval2
  ;; or do they start at the same time?
  (let ((beg1 (first interval1))
        (end1 (second interval1))
        (beg2 (first interval2))
        (end2 (second interval2)))
    (or (= beg1 beg2)
        (< beg1 beg2 end1)
        (< beg2 beg1 end2))))

;;;(defun still-in-time-interval? (interval1 interval2)
;;;  ;; is beginning of time interval2 in interval1
;;;  ;; or do they start at the same time?
;;;  (or (= (first interval2) (first interval1))
;;;      (and (> (second interval2) (first interval1))
;;;           (< (first interval2) (first interval1)))))
```

```
(defmethod simultan-notes ((timed-event rhythmic-element) &key container)
  (simultan-objects timed-event :only-type 'note :container container))

(defmethod simultan-chord ((timed-event rhythmic-element) &key container)
  (first (simultan-objects timed-event
                           :only-type 'abstract-chord
                           :container container)))

;; erstmal ineffizient - nicht ganz fertig
;;;(defmethod simultan-objects ((c container) &key (only-type t) container)
;;;  (setf container container)                ; ein ignore...
;;;  (let ((objects (list-container-objects c)))
;;;     (remove-duplicates (flat (mapcar #'(lambda (obj)
;;;                                          (simultan-objects obj :only-type only-type))
;;;                                      objects)))))

;;-----------------------------------------------------------------------------


(defmethod previous-objects ((object cm::timed-object) &optional the-container)
  ;; returns all objects surely previous to object (i.e. in threads or in threads in threads...),
  ;; nearest object first (backward order).
  (let* ((the-container (if the-container the-container
                            (cm::the-container object)))
         position result)
    (when the-container
      (setf position (object-position object the-container))
      (dotimes (i position)
        (setf result
          ;; nth-object vielleicht nicht effizient, da nth (von nth-object verwendet)
          ;; jeweils erst die ganze Liste bis zu index durchlaeuft?
          (cons (nth-object i the-container) result)))
      ;; if container of the-container is thread all objects previous to the-container
      ;; will be previous too
      (setf result
        (append result (previous-objects  the-container))))
    result))

(defmethod previous-objects ((thread cm::thread) &optional the-container)
  ;; returns all previous objects of thread if the-container of thread is thread too
  (let* ((the-container (if the-container the-container
                            (cm::the-container thread)))
         position)
    (when (typep the-container 'cm::thread) ; if the-container and its a thread
      (setf position (object-position thread the-container))
      (append
       (loop for i from (1- position) downto 0
           collect
             (reverse (list-container-objects (nth-object i the-container))))
       ;; if container of the-container is thread all objects previous to the-container
       ;; will be previous too
       (previous-objects  the-container)
      ))))


(defmethod previous-object ((object cm::timed-object) &optional container)
  ;; einfachste Loesung, aber nicht effizient, zunaechst alle Vorgaenger zu bestimmen
```

```
  (first (previous-objects object container)))

(defmethod previous-note ((object cm::timed-object))
  ;; wie previous-object: wenn first davon keine Note ist,
  ;; dann die davor. eventuell NIL.
  (let ((prevs (previous-objects object)))
    (first (remove-if-not #'(lambda (prev) (typep prev 'note))
                          prevs))))

(defmethod first-object? ((obj cm::timed-object))
  ;;Is obj first object in its container (deepest level, i.e. the-container)?
  (not (previous-objects obj)))

(defmethod second-object? ((obj cm::timed-object))
  ;;Is obj seconds object in its container (deepest level, i.e. the-container)?
  (= 1 (length (previous-objects obj))))

(defmethod all-previous-objects
    ((object cm::timed-object) &optional (only-type 'cm::timed-object))
  ;; returns all previous objects of object
  ;; i.e. also those not in the same container at deepest level
  (let* ((highest-container (first (list-object-containers object)))
         (all-objects
          (remove object
                  (filter-only-type (list-container-objects highest-container)
                                    only-type))))
    (remove-if #'(lambda (a-obj)
                   (> (slot-value a-obj 'time)
                      (slot-value object 'time)))
           (reverse
            (chronical-sort-objects all-objects)))))


(defmethod later-objects ((object cm::timed-object) &optional container)
  ;; returns all objects later to object in same container (deepest level, i.e. the-container),
  ;; beginning with the closest.
  (let* ((container
          (if container container
            (first (object-containers object))))
         position length result)
    (if container
        (progn
          (setf
              position (object-position object container)
              length (length (container-objects container)))
          (dotimes (i (- length position 1))
            (setf result
              (cons (nth-object (+ i position 1) container) result)))
          (reverse result))
      ())))

(defmethod last-object? ((obj cm::timed-object))
  ;;Is obj last object in its container, deepest level?
  (not (later-objects obj)))

(defun chronical-sort-objects (objects)
  (sort objects
        (lambda (object1 object2)
```

```
            (let ((time1 (cm::careful-slot-value object1 'time))
                  (time2 (cm::careful-slot-value object2 'time)))
              (if (and time1 time2)
                  (< time1 time2)
                (return-from chronical-sort-objects
                  (format t "~%time-slot unset, exec (run-object #!<container>)")))))))

(defmethod thread-dur ((thread cm::thread))
  ;; sums all rhythms of all events in thread
  ;; error, if thread contains not only elements or threads with elements.
  (let ((obj-list (container-objects thread))
        (dur-summe 0))
    (dolist (obj obj-list)
      (if (typep obj 'element)
          (setf dur-summe (+ (slot-value obj 'rhythm)  dur-summe))
        (setf dur-summe (+ (thread-dur obj) dur-summe))))
    dur-summe))

(defmethod thread-dur ((obj t))
  (error "thread-dur: unknown dur of ~a. dur of non threads (as merges) unspecified" obj))

;;;(defun thread-dur (thread)
;;;  (let ((note-list (container-objects thread))
;;;        (dur-summe 0))
;;;    (dolist (note note-list)
;;;      (setf dur-summe (+ (slot-value note 'rhythm)  dur-summe)))
;;;    dur-summe))

#|
;; um die gesamtlänge aller container in einem container zu erfahren:
;; geht natürlich nur für container, wo container drin sind!
;; Sollte method sein!
;; (besser waere wohl, von erstem events start,
;; und von letztem event end zu ermitteln)
(defun container-dur (container)
  (let ((container-list (container-objects container))
        (dur-summe 0))
    (dolist (container container-list)
      (setf dur-summe (+ (thread-dur container)  dur-summe)))
    dur-summe))
|#

(defmethod time-after-prev-container ((c container))
  (let* ((prev (first (previous-objects c)))
         last time rhythm)
    (if prev
        (progn
          (setf last (nth-object (1- (object-count prev)) prev))
          (if last
              (progn
                (setf time (cm::careful-slot-value last 'time)
                      rhythm (cm::careful-slot-value last 'rhythm))
                (if (and time rhythm)
                    (+ time rhythm)
                  (progn
                    (print "time or rhythm unset")
                    ())))
            ;; previous container empty
```

```
            (time-after-prev-container prev)))
        ;; c is first container (or first with content)
        0.0)))


;; !! uses vars declared in body of defcontain !!
(defmacro current-object ()
  `(nth-object i-with container))

;; !! uses vars declared in body of defcontain !!
(defmacro current-object-including-beginning ()
  `(nth-object i-without container))

;; compares two objects: which is earlier in the order of seeking?
(defmethod before-in-seeking? ((obj1 cm::timed-object) (obj2 cm::timed-object))
  (let* ((containers-obj1 (append (list-object-containers obj1) (list obj1)))
         (containers-obj2 (append (list-object-containers obj2) (list obj2)))
         (member-in-both (find-if #'(lambda (x)
                                      (member x containers-obj1)) containers-obj2))
         (compare-obj1 (object-position (second (member member-in-both containers-obj1))))
         (compare-obj2 (object-position (second (member member-in-both containers-obj2)))))
    (< compare-obj1 compare-obj2)))

;; T as BJ-Target means every obj, so its always later
(defmethod before-in-seeking? ((obj cm::timed-object) (x T))
  (if (eq x T)
      T
    (error "~a could not be compared in ~a" x #'after-in-seeking?)))
(defmethod before-in-seeking? ((x T) (obj cm::timed-object))
  (if (eq x T)
      NIL
    (error "~a could not be compared in ~a" x #'after-in-seeking?)))
(defmethod before-in-seeking? ((x T) (y T))
  (unless (and (eq x T) (eq y T))
    (error "~a or ~a could not be compared in ~a" x y #'after-in-seeking?)))

;; compares two objects: which is later in the order of seeking?
(defmethod after-in-seeking? ((obj1 cm::timed-object) (obj2 cm::timed-object))
  (let* ((containers-obj1 (append (list-object-containers obj1) (list obj1)))
         (containers-obj2 (append (list-object-containers obj2) (list obj2)))
         (member-in-both (find-if #'(lambda (x)
                                      (member x containers-obj1)) containers-obj2))
         (compare-obj1 (object-position (second (member member-in-both containers-obj1))))
         (compare-obj2 (object-position (second (member member-in-both containers-obj2)))))
    (> compare-obj1 compare-obj2)))

;; T as BJ-Target means every obj, so its always later
(defmethod after-in-seeking? ((obj cm::timed-object) (x T))
  (if (eq x T)
      NIL
    (error "~a could not be compared in ~a" x #'after-in-seeking?)))
(defmethod after-in-seeking? ((x T) (obj cm::timed-object))
  (if (eq x T)
      T
    (error "~a could not be compared in ~a" x #'after-in-seeking?)))
(defmethod after-in-seeking? ((x T) (y T))
  (unless (and (eq x T) (eq y T))
    (error "~a or ~a could not be compared in ~a" x y #'after-in-seeking?)))
```

```
;;-----------------------------------------------------------------------

;; verious helpers to clean the top-level of stella

(defun objects-without-container (&optional except)
  ;; optional are the names of the container(s) which shall be untouched
  (let ((necessary-objects '(cm::Syntaxes cm::Scratch-Score cm::Top-Level
                             cm::Pasteboard cm::Io-Streams cm::Scratch-Thread
                             cm::Toc))
        (further-necessary (if (consp except) except (list except)))
        objects result)
    (setf necessary-objects (append further-necessary necessary-objects)
          necessary-objects (mapcar #'(lambda (in)
                                        (find-object in)) necessary-objects)
          objects (remove necessary-objects
                          (cm::list-all-objects)
                          :test #'(lambda (in1 in2)
                                    (member in2 in1))))
    (dolist (o objects)
      (unless (cm::the-container o)
        (setf result (cons o result))))
    result))

(defun kill-objects (objects)
  (dolist (o objects)
    (delete-object o)
    (expunge-object o)))

(defun kill-unused (&optional except)
  (let ((objects (objects-without-container except)))
    (kill-objects objects)))
#|
(defun add-to-top-level (object)
  (setf (slot-value object 'container) (find-object 'cm::top-level)))
|#


;;-----------------------------------------------------------------------

;; ein flag-object
;;;(defclass flag ()
;;;  ((flag :initform () :initarg :flag :accessor flag)))
;;;
;;;(defmethod set-flag ((flag-object flag) content)
;;;  (setf (flag flag-object) content))
;;;
;;;(defmethod print-object ((object flag) stream)
;;;  (format stream "#<~A: ~A>"
;;;          (class-name (class-of object))
;;;          (cm::slot-value-or-default object 'flag)))

;; bj = back-jumping

(defclass bj-pointer ()
  ((pointer :accessor pointer :initarg :pointer :initform ())))

(defmethod set-pointer ((pointer bj-pointer) content)
  (let ((content (if (listp content) content (list content))))
```

```
    (setf (pointer pointer) content)))

(defmethod add-to-pointer ((pointer bj-pointer) added)
  (let ((p-list (pointer pointer))
        (added (if (listp added) added (list added))))
    (setf (pointer pointer) (append p-list added))))

(defmethod print-object ((object bj-pointer) stream)
  (format stream "#<~A: ~A>"
          (class-name (class-of object))
          (cm::slot-value-or-default object 'pointer)))

(defmethod includes-object? ((pointer bj-pointer) (obj cm::timed-object))
  (let ((p-list (pointer pointer)))
    (if p-list
        (or
         (member obj p-list)
         (member t p-list))                ; ein pointer ist T (statt eines BJ-Targets) = resultiert BT
      ;; auf keinen gezeigt = auf alle gezeigt, daraus resultiert normales BT
      t)))

(defmethod update-pointer
    ((pointer bj-pointer) (curr-obj cm::timed-object) further-objs)
  (let ((p-list (pointer pointer)))
    (if (member curr-obj p-list)
        (add-to-pointer pointer further-objs)
      (set-pointer (list curr-obj further-objs)))))
```

C.3. Die Datei *\utils.lisp

```
;        $Id: utils.lisp,v 1.10 1999/12/29 12:25:20 t Exp t $

;;*****************************************************************************
;;
;;   Copyright (c) 1999 by Torsten Anders. All rights reserved.
;;   For suggestions, comments and bug reports email to: torsten.anders@hfm.uni-weimar.de
;;
;;   Copyright (c) of Common Music 89-97, 98 Heinrich Taube. All rights reserved.
;;   Copyright (c) of Screamer 1991 Massachusetts Institute of Technology. All rights reserved.
;;                             1992, 1993 University of Pennsylvania. All rights reserved.
;;                             1993 University of Toronto. All rights reserved.
;;
;;   This program is free software; you can redistribute it and/or modify
;;   it under the terms of the GNU General Public License as published by
;;   the Free Software Foundation; either version 2 of the License, or
;;   (at your option) any later version.
;;
;;*****************************************************************************

(in-package :arno)

(defvar *utils*                                ; folgende Symbole werden exportiert
    '(flat mat-trans first-n atom-position filter-only-type filter-only-types
      filter-unless-typeremove-list rescale-list
      let*-when
```

```
      either-bj a-member-of-bj an-integer-between-bj
      a-shuffled-expr  an-octave-of append-octaves put-ratios-in-one-octave
      in-range? mac map-math round-rhythm
      freq2keynum invert-number))

;; list
(defun flat (input)
  (reverse (flat-aux1 input)))

(defun flat-aux1 (in)
  (if (null in) nil
    (if (atom in) (list in)
      (append (flat-aux1 (rest in)) (flat-aux1 (first in))))))

(defun mat-trans (in-list)
  ;; um eine Liste der Form ((a1 a2 a3) (b1 b2 b3) (c1 c2 c3) ...)
  ;; umzuformen in ((a1 b1 c1 ...) (a2 b2 c2 ...) (a3 b3 c3 ...))
  (apply #'mapcar #'(lambda (&rest all)
                      all)
         in-list))

(defun first-n (in-list n)
  (if (< (length in-list) n)
      in-list
    (butlast in-list (- (length in-list) n))))

(defun atom-position (atom in-list)
  ;; returns zero based (first) position of atom in list
  (let ((list-length (length in-list))
        (member-out (member atom in-list)))
    (when member-out
      (- list-length (length member-out)))))

(defun filter-only-type (in-list type)
  (remove-if-not #'(lambda (in)
                     (typep in type))
                 in-list))

(defun filter-only-types (in-list type-list)
  (remove-if-not
   #'(lambda (in)
       (eval (cons 'or
                   (mapcar
                    #'(lambda (type)
                        (typep in type))
                    type-list))))
   in-list))

(defun filter-unless-type (in-list type-to-remove)
  (remove-if #'(lambda (in)
                 (typep in type-to-remove))
             in-list))

#|
(defun types? (in type-list)
  ;; returns in, when in is typep of one type of type-list
  (when (eval (cons 'or
                    (mapcar #'(lambda (type)
```

```
                                (typep in type))
                              type-list)))
    in))
|#

(defun remove-list (elements-to-remove in-list)
  ;; entfernt alle elemente von elements-to-remove aus in-list
  (if elements-to-remove
      (remove-list (rest elements-to-remove)
                   (remove (first elements-to-remove) in-list))
    in-list))

(defun rescale-list (list new-min new-max)
  ;; skaliert geg. liste in den Wertebereich new-min bis new-max
  ;; (weil new-min=0 sein muss nur vorlaeufige Version)
  (let* ((old-min (apply #'min list))
         (old-max (apply #'max list)))
    (mapcar #'(lambda (val) (rescale val old-min old-max new-min new-max)) list)))


;; aus OM kopiert (...)
(defun x->dx (list)
  ;; computes a list of intervals from a list of points.
  (loop for x in list
        for y in (rest list)
        collect (- y x)) )

(defun dx->x (start list)
  ;; computes a list of points from a list of intervals and a <start> point
  (cons start (loop for dx in list
                    sum dx into thesum
                    collect (+ start thesum)) ))

;;-----------------------------------------------------------------------
;; Predicates

(defun true? (in)
  ;; gibt nur T oder NIL zurueck, noetig wegen screamer...
  (if in t ()))


(defmacro all-true? (to-test predicate-list)
  ;; returns true, if to-test fulfills all predicates of predicate-list
  ;; evaluiert leider nochmal explizit:
  ;; wie kann ich den Wert einer Variable (liste) mit AND testen?
  (let ((result (gensym)))
    `(let ((,result))
       (dolist (predicate ,predicate-list)
         (setf ,result
           (append ,result
                   (list (funcall predicate ,to-test)))))
       (when *write-statistics* (update-defcontain-statistics ,result ,predicate-list))
       (and-fun ,result))))

(defmacro non-true? (to-test predicate-list)
  ;; returns true, if to-test fulfills all predicates of predicate-list
  ;; evaluiert leider nochmal explizit:
  ;; wie kann ich den Wert einer Variable (liste) mit AND testen?
```

```
  (let ((result (gensym)))
    `(let ((,result))
       (dolist (predicate ,predicate-list)
         (setf ,result
           (append ,result
                   (list (not (funcall predicate ,to-test))))))
       (when *write-statistics* (update-defcontain-statistics ,result ,predicate-list))
       (and-fun ,result))))

;;-----------------------------------------------------------------------
;; closures

(defmacro let*-when (binds &body body)
  ;; wie let*, aber testet nach jeder Zuweisung, ob diese non-NIL
  ;; ist eine Bindung NIL, ist ganzer Ausdruck NIL, sonst Body
  ;; ("avoid-constraint" dann passed)
  ;; (ziteirt aus Graham, "On Lisp", dort when-binds* genannt)
  (if (null binds)
      `(progn ,@body)
    `(let (,(car binds))
       (if ,(caar binds)
           (let*-when ,(rest binds) ,@body)
         ))))

;;-----------------------------------------------------------------------

#|
(defun one-=? (value list)
  ;; returns true, if one of the elements of list = value
  ;; entspr. (find value list :test #'=)
  (let ((boolean-list (mapcar
                       #'(lambda (in) (= value in))
                       list)))
     (eval
      (cons 'or boolean-list))))

(defmacro one-eq? (value list)
  ;; returns true, if one of the elements of list eq value
  ;; entsp. (find value list)
  `(let ((boolean-list (mapcar
                       #'(lambda (in) (eq ,value in))
                       ,list)))
     (eval
      (cons 'or boolean-list))))

(defmacro map-all-true? (to-test-list predicate-list)
  ;; returns true, if all elements of to-test-list fulfill
  ;; all predicates of predicate-list
  `(let ((result '(and)))
     (setf result
       (append result
               (mapcar #'(lambda (in)
                           (all-true? in ,predicate-list))
                       ,to-test-list)))
     (eval result) ))

(defmacro one-true? (to-test predicate-list)
  `(let ((result '(or)))
```

```
      (dolist (predicate ,predicate-list)
        (setf result
          (cons
           (list predicate ,to-test)
           result)))
      (eval (reverse result))))
|#


;;-----------------------------------------------------------------------------

;; nondeterministic utils: muessen defuns (oder macros) sein, keine defmethods!

(defmacro a-shuffled-expr (nondeterm-expr)
  ;; returns poss. values of nondeterm-expr in shuffled order
  `(let ((shuffled (shuffle (all-values ,nondeterm-expr) :copy NIL)))
     ;; cm::shuffle returns as default a copy of input
     (a-member-of shuffled)))

(defmacro shuffled-or-not (shuffle? nondeterm-expr)
  ;; returns poss. values of nondeterm-expr in un-shuffled or shuffled order
  ;; depending on shuffle?
  `(if ,shuffle?
       (a-shuffled-expr ,nondeterm-expr)
     ,nondeterm-expr))



;; Allg. Syntax fuer nondeterm -testing: Schreibweise wie gehabt
;; mit angehaengten curr-obj und pointer-obj.
;; Bei either-testing diese Reihenfolge vertauscht (wegen &rest)

(defun either-testing (curr-obj pointer-obj &rest possibilities)
  (if (and
       (includes-object? pointer-obj curr-obj) possibilities)
      (either (first possibilities)
              (apply-nondeterministic
               #'either-testing (append (list curr-obj pointer-obj)
                                        (rest possibilities))))
    (progn
      (unset-object curr-obj)
      (fail))))

(defmacro either-bj (&rest possibilities)
  `(either-testing (current-object) *bj-pointer* ,@possibilities))

(defun an-integer-between-testing (low high curr-obj pointer-obj)
  (if (or
       (not (includes-object? pointer-obj curr-obj))
       (> low high))
      (progn
        (unset-object curr-obj)
        (fail))
    (either low
            (an-integer-between-testing (1+ low) high curr-obj pointer-obj))))

(defmacro an-integer-between-bj (low high)
```

```
  `(an-integer-between-testing ,low ,high (current-object) *bj-pointer*))

(defun a-member-of-testing (list curr-obj pointer-obj)
  (if (and
       (includes-object? pointer-obj curr-obj) list)
      (either (first list)
              (a-member-of-testing (rest list) curr-obj pointer-obj))
    (progn
      (unset-object curr-obj)
      (fail))))

(defmacro a-member-of-bj (list)
  `(a-member-of-testing ,list (current-object) *bj-pointer*))

;;-----------------------------------------------------------------------------


(defun an-octave-of (ratio &optional (range 4))
  (let (factor-list aux octaves)
    (dotimes (i range)
      (setf aux (expt 2 (1+ i))
            factor-list (cons aux factor-list)
            factor-list (cons (/ 1 aux) factor-list)))
    (setf factor-list (cons 1 (reverse factor-list))
          octaves
          (mapcar #'(lambda (x) (* ratio x)) factor-list))
    (a-member-of  octaves)))

(defun append-octaves (ratios  &optional (octave-range 4))
  ;; haengt determ. Oktavierungen aller ratios vorn und hinten dran
  ;; wenn das gutgehen soll, muessen alle ratios in einer Oktave liegen...
  (let (factors aux)
    (dotimes (i octave-range)
      (setf aux (expt 2 (1+ i))
            factors (cons aux factors)
            factors (cons (/ 1 aux) factors)))
    (setf factors (cons 1 factors)
          factors (sort factors #'<))
    (flat
     (mapcar #'(lambda (factor)
                 (mapcar #'(lambda (ratio)
                             (* ratio factor)) ratios)) factors))))

(defun put-ratios-in-one-octave (ratios)
  ;; erste Note bleibt in gleicher Oktave und tiefster Ton,
  ;; alle anderen werden, wenn noetig, in diese Oktave transponiert
  (let ((reverence (first ratios)))
    (mapcar
     #'(lambda (ratio)
         (let ((exp (log (/ reverence ratio) 2)))
           (setf exp (ceiling exp))
           (* ratio (expt 2 exp))))
     ratios)))

(defun put-ratios-in-given-octave (ratios reverence)
  ;; ratios in Oktav, reverence in Mitte
  (let* ((tritone (expt (expt 2 1/12) 6))
         (reverence (/ reverence tritone)))
```

```
      (mapcar
       #'(lambda (ratio)
           (let ((exp (log (/ reverence ratio) 2)))
             (setf exp (ceiling exp))
             (* ratio (expt 2 exp))))
       ratios)))


;;-----------------------------------------------------------------------------

;; sonstige utils


(defun in-range? (freq-to-test freq &optional (range-ratio 5/4))
  ;; Ist freq-to-test in range-ratio von freq?
  (and
   (> freq-to-test (/ freq range-ratio))
   (< freq-to-test (* freq range-ratio))))

(defmacro mac (expr)
  ;; short-hand to macroexpand macros
  `(pprint (macroexpand-1 ',expr)))

(defmethod map-math ((fun function) (atom number) (liste list))
  ;; "mapt" uebergebene Funktion mit uebergebenen Atom und Liste
  (mapcar #'(lambda (in) (funcall fun atom in)) liste))

(defmethod map-math ((fun function) (liste list) (atom number))
  (mapcar #'(lambda (in) (funcall fun in atom)) liste))

(defun round-rhythm (rhythm-val smallest-val)
  ;; rundet rhythm-val auf n Vielfache von smallest-val
  (* smallest-val (round (/ rhythm-val smallest-val))))

(defun and-fun (vals)
  ;; ein AND, dass als arg Liste oder ein Symbol erwartet, dessen Wert Liste ist
  ;; ist das als Macro sinnvoller?
  (case (length vals)
    (0 t)
    (1 (first vals))
    (t (if (first vals)
           (and-fun (rest vals))))))

;; um freqs in keynum umzurechnen, auch floats moegl.
(defmethod freq2keynum ((ref number))
  (let ((scale cm::*chromatic-scale*))
    (values (log (/ ref (slot-value scale 'cm::lowest))
                 (slot-value scale 'cm::interval-ratio)))))

(defmethod invert-number ((number number) (point number))
  ;; eine Verallgemeinerung von cm::invert
  (+ point (* (- number point) -1)))
```

C.4. Die Datei *\envelope.lisp

```
;        $Id: envelope.lisp,v 1.10 1999/12/29 12:24:59 t Exp $

;;*****************************************************************************
;;
;;   Copyright (c) 1999 by Torsten Anders. All rights reserved.
;;   For suggestions, comments and bug reports email to: torsten.anders@hfm.uni-weimar.de
;;
;;   Copyright (c) of Common Music 89-97, 98 Heinrich Taube. All rights reserved.
;;   Copyright (c) of Screamer 1991 Massachusetts Institute of Technology. All rights reserved.
;;                             1992, 1993 University of Pennsylvania. All rights reserved.
;;                             1993 University of Toronto. All rights reserved.
;;
;;   This program is free software; you can redistribute it and/or modify
;;   it under the terms of the GNU General Public License as published by
;;   the Free Software Foundation; either version 2 of the License, or
;;   (at your option) any later version.
;;
;;*****************************************************************************

(in-package :arno)

(defvar *envelope*
    '(env-value lower-env upper-env append-envs extract-env
      reverse-env invert-env scale-env
      map-env-values map-env-on-object env2list))
#|
Eine einfache env hat die Form ((zeit0 val0) ... (zeit1 val1))
mit zeit0 = 0 (env-anfang) und zeit1 = 1 (env-ende).
Val kann beliebige Werte annehmen.

Versch. Varianten:

Es werden gleich zwei envs ausgedrueckt, eine obere und eine untere:
((zeit0 val0a val0b) ... (zeit1 val1a val1b)), extrahieren mit lower-env oder
upper-env noetig.

In env sind vals (auch Zeitwerte) als expr. moegl und in expr. koennen auch
nondeterministische Ausdruecke vorkommen. Aber in beiden Faellen entwas andere Syntax noetig:
(list (list zeit0 (+ variable 1)) ... (list zeit1 val1))
(list (list zeit0 (+ variable (an-integer-between 0 5)))
      (list zeit1 val1))
Wird mit eval-env evaliert.
|#

;; Aenderung: entscheidend ist der env-val beim Startpunkt der Note, d.h. z.B. wenn
;; i = length (eins nach letztem Wert) ist dies letzter Wert in env
(defun env-value (i length env)
  ;; i is zero-based, length one-based
  ;; Fall length=0 ergibt ersten env-val
  (let ((envelope (if (listp (first env))
                      (flat (eval-env env))
                    env)))                ; test nur, ob erstes element liste oder atom
  (if (= 1 length)
      (second (first env))
    (interp  (/ i length) envelope))))
```

```
(defun eval-env (env &optional result)
  ;; evaluieren funktioniert nur mit der Form
  ;; (list (list 0 (+ 1 2)) (list 1 1))
  ;; -> kann eval-env dann nicht ganz weg ?
  ;; envs ohne expr aber auch in Form '((0 0) (1 1)) moeglich
  (if env
      (let ((time (first (first env)))
            (val (second (first env))))
        (eval-env (rest env) (append result (list (list time val)))))
    result))

;;;(defun env-value-expl (i env &key (base 2))
;;;  ;; buged !!
;;;
;;;  ;; expl laesst sich auf jedes env-seg einzeln anwenden,
;;;  ;; brauche also upper und lower vals von env-seg und
;;;  ;; muss i in dieses seg transferieren
;;;  ;; (was mit length?)
;;;  (let* ((lower-pair
;;;          (if (= i 1) (second (reverse env))
;;;            (find i (reverse env) :test #'(lambda (val pair) (<= (first pair) val)))))
;;;         (upper-pair
;;;          (if (= i 0) (second env)
;;;            (find i env :test #'(lambda (val pair) (>= (first pair) val)))))
;;;         (expl-power (scaled-time-whole2part i (first lower-pair) (first upper-pair))))
;;;    (expl expl-power :base base)))

(defun lower-env (in &optional result)
  (if (not in)
      result
    (let ((x (first (first in)))
          (y (second (first in))))
      (setf result (append result (list (list x y))))
      (lower-env (rest in) result))))

(defun upper-env (in &optional result)
  (if (not in)
      result
    (let ((x (first (first in)))
          ;; wenn es einen dritten wert gibt, nimm diesen, sonst den zweiten
          (y (if (third (first in))
                 (third (first in))
               (second (first in)))))
      (setf result (append result (list (list x y))))
      (upper-env (rest in) result))))

;;-----------------------------------------------------------------------------

(defun curr-env-part (startindex number-part number-all env)
  ;; startindex zerobased,  number-part and number-all onebased
  ;; end-wert des Vorgaengers = Start-Wert des Nachfolger
  (let* ((start-time (/ startindex  number-all))
         (end-time (/ (+ startindex number-part)  number-all))
         (start-val (interp start-time (flat (eval-env env))))
         (end-val (interp end-time (flat (eval-env env))))
         (pairs-in-time
          (remove-if-not #'(lambda (env-pair)
                             (<= start-time (first env-pair) end-time)) env)))
```

```
    (setf pairs-in-time (mapcar #'(lambda (pair)
                                    (list (scaled-time-whole2part (first pair)
                                                                  start-time end-time)
                                          (second pair)))
                                pairs-in-time)
          pairs-in-time (append (list (list 0 start-val))
                                pairs-in-time
                                (list (list 1 end-val))))
    (remove-duplicates pairs-in-time :TEST #'(lambda (pair1 pair2)
                                               (= (first pair1)
                                                  (first pair2))))))


(defun scaled-time-whole2part (time-val start-time end-time)
  ;; ueber Gleichungssystem eine lineare Funktion ermitteln (y=ax+b),
  ;; mit der skalierte time-val ermittelbar sind
  ;; whole-time2part-time
  ;; Was ist time-val aus ganzer env-dur in env-dur zw. start-time (dann 0)
  ;; und end-time (dann 1)
  (let* ((a (/ 1 (- end-time start-time)))
         (b (- (* a start-time))))
    (+ (* a time-val) b)))

(defun scaled-time-part2whole (time-val start-time end-time)
  ;; x=(y-b)/a
  ;; part-time2whole-time
  ;; Was ist time-val aus teil-env-dur in env-dur zw. start-time (in teil-env 0)
  ;; und end-time (in teil-env 1)
  (let* ((a (/ 1 (- end-time start-time)))
         (b (- (* a start-time))))
    (/ (- time-val b) a)))

;;-----------------------------------------------------------------------------

;; cm::rescale tuts mit einzelnen Werten
;;;(defun scale (list new-max)
;;;  ;; skaliert geg. liste in den Wertebereich 0 bis new-max
;;;  ;; (weil new-min=0 sein muss nur vorlaeufige Version)
;;;  (let* ((new-min 0)
;;;         (old-min (apply #'min list))
;;;         (old-max (apply #'max list))
;;;         (add (- new-min old-min))
;;;         (mult (/ new-max (+ old-max new-min (- old-min)))))
;;;    (map-math #'* mult
;;;              (map-math #'+ add

(defun check-env-times (env)
  ;; nur eine Sicherheitsfkt., das Zeitwerte nicht aus Wertebereich 0-1 fallen
  (if (null env)
      nil
    (let ((times (first (mat-trans env)))
          (vals (second (mat-trans env))))
      (mat-trans
       (list (rescale-list times 0 1) vals)))))

;;-----------------------------------------------------------------------------

;; Aneinanderstossende env-ende und env-anf koennen verschiedene vals haben:
```

```
;; interp gibt immer nur den zweiten Wert zurueck.
;; Soll time-val von erstem Wert leicht verkuerzt werden?

(defun append-envs (&rest envs)
  (check-env-times (apply #'append (append-envs-aux envs))))

(defun append-envs-aux (envs-list &optional (time-offset 0))
  (if (endp envs-list)
      nil
    (cons (add-to-all-env-times (first envs-list) time-offset)
          (append-envs-aux (rest envs-list) (1+ time-offset)))))

(defun append-envs-obj-aux (envs-list number-list &optional (time-offset 0))
  ;; number-list enthaelt die number fuer die jeweilige env
  (if (endp envs-list)
      nil
    (let* ((number (first number-list))
           (curr-env (mult-to-all-env-times (first envs-list) number)))
      (cons (add-to-all-env-times curr-env time-offset)
            (append-envs-obj-aux (rest envs-list) (rest number-list) (+ time-offset number) )))))


(defun add-to-all-env-times (env add)
  ;; aux fuer append-envs
  (mapcar #'(lambda (pair)
              (list
               (+ (first pair) add)
               (second pair)))
          env))

(defun mult-to-all-env-times (env factor)
  ;; aux fuer append-envs
  (mapcar #'(lambda (pair)
              (list
               (* (first pair) factor)
               (second pair)))
          env))

;;-----------------------------------------------------------------------------

(defun reverse-env (env)
  (mapcar #'(lambda (pair)
              (cons (time-reziproke (first pair))
                    (rest pair)))
          (reverse env)))

(defun invert-env (env &optional reference)
  ;; geht von einfacher env aus: ((t1 val1) (t2 val2) ... (tn valn))
  ;; ohne explizite Angabe des "Spiegeliungspkts" wird dieser aus Durchschnitt der envs vals gebildet:
  (let ((intern-ref (if reference
                        reference

                      (let ((vals (second (mat-trans env))))
                        (/ (apply #'+ vals) (length vals))))))
    (mapcar #'(lambda (pair)
                (list (first pair)
                      (invert-number (second pair) intern-ref)))
            env)))
```

```
(defun scale-env-values (env mix max)
  ;; um eine env umzuformen
  ;; (war mal scale-env!)
  (if (null env)
      nil
    (let* ((times (first (mat-trans env)))
           (vals (second (mat-trans env))))
      (mat-trans
       (list times (rescale-list vals mix max))))))

(defun scale-env (env)
  ;; skaliert Zeiten einer env so, dass sie immer zwischen 0 und 1 liegen
  ;; (war mal wie scale-env-values!)
  (let ((min 0)
        (max 1))
    (if (null env)
        nil
      (let* ((times (first (mat-trans env)))
             (vals (second (mat-trans env))))
        (mat-trans
         (list (rescale-list times min max) vals))))))

;;-----------------------------------------------------------------------------

(defmethod extract-env ((container container) (parameter symbol))
  ;; bildet env aus geg. Parameter der Events in Container
  ;; dazu muessen alle Object in Container diesen Slot enthalten.
  (let ((length (object-count container))
        curr-val result)
    (dotimes (i length)
      (setf curr-val (cm::careful-slot-value (nth-object i container) parameter)
            result (cons (list (/ i length) curr-val) result)))
    ;; statt Extrapolation letzter Wert am Schluss nochmal:
    (setf result (cons (list 1 curr-val) result))
    (reverse result)))

(defun map-env-values (env expr)
  ;; applys expr to all values (i.e. not times) of env
  (mapcar #'(lambda (pair)
              (list (first pair)
                    (apply expr (rest pair)))) env))

;; eigentlich ein set, kein map und on container nicht object
(defmethod map-env-on-object ((c container) (env cons) (slot symbol))
  ;; maps ueber alle env an slot elemente in container, nur ein level
  (let ((number (object-count c)))
    (dotimes (i number)
      (let ((val (env-value i number env)))
        (set-object (nth-object i c)
                    slot val)))
    c))

(defun env2list (env length)
  (let (result)
    (dotimes (i length)
      (push (env-value i length  env)
            result))
```

```
    (reverse result)))
```

C.5. Die Datei *\envelope-object.lisp

```
;        $Id: envelope-object.lisp,v 1.12 1999/12/29 12:24:48 t Exp $

;;*****************************************************************************
;;
;;   Copyright (c) 1999 by Torsten Anders. All rights reserved.
;;   For suggestions, comments and bug reports email to: torsten.anders@hfm.uni-weimar.de
;;
;;   Copyright (c) of Common Music 89-97, 98 Heinrich Taube. All rights reserved.
;;   Copyright (c) of Screamer 1991 Massachusetts Institute of Technology. All rights reserved.
;;                             1992, 1993 University of Pennsylvania. All rights reserved.
;;                             1993 University of Toronto. All rights reserved.
;;
;;   This program is free software; you can redistribute it and/or modify
;;   it under the terms of the GNU General Public License as published by
;;   the Free Software Foundation; either version 2 of the License, or
;;   (at your option) any later version.
;;
;;*****************************************************************************

(in-package :arno)

(defvar *envelope-object*
    '(envs-object set-number set-number-env set-rhythm-env ; symbole werden exportiert
      get-number-value get-rhythm-value envs-dur set-envs-dur number number-env rhythm-env
      simple-note-envs set-dur-env set-amp-env set-pitch-env
      get-dur-value get-amp-value get-pitch-value dur-env amp-env pitch-env
      set-object-as-envs-value append-simple-note-envs curr-envs-part
      set-envs-as-envs-value variate-envs-as-envs-value
      variate-envs-static variate-envs-number time-warp-envs amplify-envs
      transp-envs scale-envs-pitch reverse-envs invert-envs variate-envs-with-envs
      envs2thread thread2envs))

;; eigenes Object, das envs fuer jeden Parameter enthaelt
;; envs in env-object genuegen der selben Syntax wie einfache env
;; (siehe envelope.lisp)
;; Anzahl ist extra Slot

;; um quasi Motive ueber die Grenzen von Time-slices ragen zu lassen:
;; wird set-object-as-envs-value angewendet auf Container,
;; wird teil-env auf jedes Object in Container angewendet.
;; Wann aber passiert dann Test bei nondeterm.?

(defclass envs-object ()
  ;; beim Initialisieren kein automatisches scale-env
  ((number :accessor number :initarg :number)
   ;; number-env fuer Verwendeung als envs-offset-object
   (number-env :accessor number-env :initarg :number-env :initform '((0 1) (1 1)))
   (rhythm-env :accessor rhythm-env :initarg :rhythm-env :initform '((0 1) (1 1)))))

(defmethod print-object ((object envs-object) stream)
  (format stream "#<~A: ~A ~17T~A >"
          (class-name (class-of object))
```

```
          (cm::slot-value-or-default object 'number)
          (cm::slot-value-or-default object 'rhythm-env)))

(defmethod set-number  ((obj envs-object) (number number))
  (setf (number obj) (round number))
  obj)

(defmethod set-number-env ((obj envs-object) env)
  (setf (number-env obj) (check-env-times env))
  obj)

(defmethod set-rhythm-env ((obj envs-object) env)
  (setf (rhythm-env obj) (check-env-times env))
  obj)

(defmethod get-number-value ((envs envs-object) (i integer))
  ;; get rhythm of ith note of envs (zero-based)
  (env-value i (number envs) (number-env envs)))

(defmethod get-rhythm-value ((envs envs-object) (i integer))
  ;; get rhythm of ith note of envs (zero-based)
  (env-value i (number envs) (rhythm-env envs)))

;;benoetigt?
;;;(defmethod set-object-as-envs-value ((obj cm::timed-object) (env-obj envs-object)
;;;                                &optional (i nil))
;;;  (let ((i (if i i (object-position obj))))
;;;    (set-object obj
;;;                'rhythm (get-rhythm-value env-obj i))))

;; Laenge (Dauer) einer Stimme determiniert durch Anzahl der Noten (number) und Dauern (rhythm-env)
;; zwei Methoden die env-obj variieren zu geg. Gesamtdauer durch
;;  - Variation der Anzahl
;;  - Skalierung der rhythm-env

(defmethod envs-dur ((envs envs-object))
  (let ((dur 0))
    (dotimes (i (number envs))
      (setf dur (+ dur (get-rhythm-value envs i))))
    dur))

(defmethod set-envs-dur ((envs envs-object) (dur number)
                                    &key (per 'number) (min-or-max 'min))
  ;; keine Variation, envs-obj selbst wird geaendert
  ;; arg min-or-max nimcht bei :per 'rhythm
  (case per
    (number
     (case min-or-max
       (min
        (do ((duration (envs-dur envs) (envs-dur envs)))
            ((> duration dur) envs)
          (set-number envs (1+ (number envs)))))
       (max
        (do ((duration (envs-dur envs) (envs-dur envs)))
            ((< duration dur) envs)
          (set-number envs (1- (number envs)))))))
    (rhythm
     (let* ((curr-dur (envs-dur envs))
```

```
            (dur-factor (/ dur curr-dur))
            (new-rhythm-env (mapcar #'(lambda (pair)
                                        (list (first pair)
                                              (* (second pair) dur-factor)))
                                    (rhythm-env envs))))
       (set-rhythm-env envs new-rhythm-env)))))

;; offset-envs haben keine eigene dauer, aber factor. Wenn in envs, die sie
;; beeinflussen soll dur=1 und number=1 deren Dauer mit offset-envs  setzbar.
;; Oder envs-object wird zs. mit envs-offset-env als Parameter uebergeben
;;;(defmethod set-offset-envs-dur
;;;    ((offset-envs offset-envs-object) (envs envs-object) (dur number))
;;;  ())


;;-----------------------------------------------------------------------------
;; envs mit amp und pitch: simple-note-envs

(defclass simple-note-envs (envs-object)
  ;; beim Initialisieren kein automatisches scale-env
  ;; pitch-env (statt note-env), da beim Interpolieren floats entstehen
  ;; (werden von cm als pitch in Hz interpretiert)
  ;; d.h. alle Werte muessen auch als float geg. werden!
  ((dur-env :accessor dur-env :initarg :dur-env :initform NIL)
   (amp-env :accessor amp-env :initarg :amp-env :initform '((0 1) (1 1)))
   (pitch-env :accessor pitch-env :initarg :pitch-env :initform '((0 1) (1 1)))))

(defmethod print-object ((object simple-note-envs) stream)
  (format stream "#<~A: ~A ~%~17T~A ~%~17T~A ~%~17T~A ~%~17T~A >"
          (class-name (class-of object))
          (cm::slot-value-or-default object 'number)
          (cm::slot-value-or-default object 'rhythm-env)
          (cm::slot-value-or-default object 'dur-env)
          (cm::slot-value-or-default object 'amp-env)
          (cm::slot-value-or-default object 'pitch-env)))

(defmethod set-dur-env ((obj simple-note-envs)  env)
  (setf (dur-env obj) (check-env-times env))
  obj)

(defmethod set-amp-env ((obj simple-note-envs)  env)
  (setf (amp-env obj) (check-env-times env))
  obj)

(defmethod set-pitch-env ((obj simple-note-envs)  env)
  (setf (pitch-env obj) (check-env-times env))
  obj)


(defmethod get-dur-value ((envs simple-note-envs) (i integer))
  ;; get rhythm of ith note of envs (zero-based)
  (env-value i (number envs) (dur-env envs)))

(defmethod get-amp-value ((envs simple-note-envs) (i integer))
  ;; get rhythm of ith note of envs (zero-based)
  (env-value i (number envs) (amp-env envs)))

(defmethod get-pitch-value ((envs simple-note-envs) (i integer))
```

```
  ;; get rhythm of ith note of envs (zero-based)
  (env-value i (number envs) (pitch-env envs)))

(defmethod set-object-as-envs-value ((obj midi-note) (env-obj simple-note-envs)
                                     &optional (i nil))
  ;; amplitude (0-1!) -> velos, note (pitch!) -> degree
  (let* ((i (if i i (object-position obj)))
         (dur-val (if (dur-env env-obj)
                      (get-dur-value env-obj i)
                    (get-rhythm-value env-obj i))))
    (set-object obj
                'cm::rhythm (get-rhythm-value env-obj i)
                'cm::duration dur-val
                'cm::amplitude (amplitude (get-amp-value  env-obj i)
                                          :softest 0 :loudest 127)
                'cm::note (note (get-pitch-value env-obj i))) ))

(defmethod set-object-as-envs-value ((obj csound-note) (env-obj simple-note-envs)
                                     &optional (i nil))
  ;; amplitude (0-1!) -> velos, note (pitch!) -> degree
  (let* ((i (if i i (object-position obj)))
         (dur-val (if (dur-env env-obj)
                      (get-dur-value env-obj i)
                    (get-rhythm-value env-obj i))))
    (set-object obj
                'cm::rhythm (get-rhythm-value env-obj i)
                'cm::dur dur-val
                'cm::amp (amplitude (get-amp-value  env-obj i)
                                    :softest 0 :loudest 20000)
                'cm::freq (get-pitch-value env-obj i))))

(defun append-simple-note-envs (&rest all-envs)
  ;; jede einzelne env durch eigene number verschoben!
  (let* ((new-obj (copy-object (first all-envs)))
         (all-numbers (mapcar #'number all-envs))
         (number-sum (apply #'+ all-numbers))
         ;; besser waer, z.B. auf Endg. -env zu testen?
         (slot-list (remove 'number (CLOS::OBJECT-INSTANCE-SLOT-NAMES new-obj))))
    (set-number new-obj number-sum)
    (dolist (slot slot-list)
      (let* ((all-envs-of-slot (mapcar slot all-envs))
             (whole-env-of-slot
              ;; scale env scaliert Zeiten (von 0 bis 1)
              (scale-env (apply #'append (append-envs-obj-aux all-envs-of-slot all-numbers)))
              ))
        (set-object new-obj slot whole-env-of-slot)))
    new-obj))

(defmethod curr-envs-part ((start-index integer) (number-part integer) (envs simple-note-envs))
  ;; noch number einbaun: geg. startindex, anzahl
  (let ((envs-part (copy-object envs))
        (number-all (number envs)))
    (set-number envs-part number-part)
    (set-rhythm-env envs-part (curr-env-part start-index number-part number-all (rhythm-env envs)))
    (set-dur-env envs-part (curr-env-part start-index number-part number-all (dur-env envs)))
    (set-amp-env envs-part (curr-env-part start-index number-part number-all (amp-env envs)))
    (set-pitch-env envs-part (curr-env-part start-index number-part number-all (pitch-env envs)))
    envs-part))
```

```
(defmethod set-envs-as-envs-value
    ((envs-obj simple-note-envs) (offset-envs simple-note-envs) (i integer))
  ;; keine Variation, setzt Werte von envs-obj direkt
  ;; amplitude (0-1!) -> velos, note (pitch!) -> degree.
  ;; Leider muss i (i-with)  direkt uebergeben werden.
  ;; Natuerlich Funktion eigentlich kein set sondern ein Scaling.
  (let ((length (number offset-envs))
        (envs-dur-env (if (dur-env envs-obj)
                          (dur-env envs-obj)
                        (rhythm-env envs-obj)))
        (offset-dur-env (if (dur-env offset-envs)
                            (dur-env offset-envs)
                          (rhythm-env offset-envs))))
    (set-number envs-obj
                (* (number envs-obj)
                   (env-value i length (number-env offset-envs))))
    (set-rhythm-env envs-obj
                    (map-env-values (rhythm-env envs-obj)
                                    (lambda (x)
                                      (* x (env-value i length (rhythm-env offset-envs))))))
    (set-dur-env envs-obj
                    (map-env-values envs-dur-env
                                    (lambda (x)
                                      (* x (env-value i length offset-dur-env)))))
    (set-amp-env envs-obj
                    (map-env-values (amp-env envs-obj)
                                    (lambda (x)
                                      (* x (env-value i length (amp-env offset-envs))))))
    (set-pitch-env envs-obj
                    (map-env-values (pitch-env envs-obj)
                                    (lambda (x)
                                      (* x (env-value i length (pitch-env offset-envs))))))
    envs-obj))

(defmethod variate-envs-as-envs-value
    ((envs-obj simple-note-envs) (offset-envs simple-note-envs) (i integer))
  ;; gibt variierte Kopie von envs-obj zurueck
  ;; amplitude (0-1!) -> velos, note (pitch!) -> degree
  (let ((variation (copy-object envs-obj)))
    (set-envs-as-envs-value variation offset-envs i)))

;;-----------------------------------------------------------------------
;; Transformationen, Variationen von envs-objects:
;; zurueckgegeben wird ein Kopie des geg. Objektes (ausser :copy? ist NIL)

(defmethod variate-envs-static
    ((env-obj envs-object) (env-slot symbol)
     (var-fun function) (var-val number) &key (copy? t))
  (let* ((result-envs (if copy? (copy-object env-obj) env-obj))
         (to-variate (funcall env-slot result-envs))
         variated)
    (setf variated
      (if (atom to-variate)
          (funcall var-fun to-variate var-val) ; um z.B. number zu variiren
        (mapcar #'(lambda (pair)
                    (list (first pair)
```

```
                          (funcall var-fun (second pair)  var-val)))
                to-variate)))
      (set-object result-envs env-slot variated)
      result-envs))

;;;(defmethod variate-envs-number ((envs envs-object) (ratio number))
;;;  (let* ((envs-copy (copy-object envs)))
;;;    (set-number envs-copy (* (number envs-copy) ratio))
;;;    envs-copy))

(defmethod variate-envs-number ((envs envs-object) (ratio number) &key (copy? t))
   (variate-envs-static envs 'number #'* ratio :copy? copy?))

(defmethod time-warp-envs ((env envs-object) (ratio number) &key (copy? t))
  (variate-envs-static
   (variate-envs-static env 'rhythm-env #'* ratio :copy? copy?)
   'dur-env #'* ratio :copy? copy?))

;; Vorsicht, immer 0 < amp < 1 ! (indem Variation ungeprueft nie > 1)
(defmethod amplify-envs ((env simple-note-envs) (ratio number) &key (copy? t))
  (variate-envs-static env 'amp-env #'* ratio :copy? copy?))

(defmethod transp-envs ((env simple-note-envs) (ratio number) &key (copy? t))
  (variate-envs-static env 'pitch-env #'* ratio :copy? copy?))

(defmethod scale-envs-pitch
    ((envs simple-note-envs) (newmin number) (newmax number))
  (let ((min (apply #'min (second (mat-trans (pitch-env envs)))))
        (max (apply #'max (second (mat-trans (pitch-env envs))))))
    (set-pitch-env envs (scale-env-vals (pitch-env envs)
                                        min max newmin newmax))))


(defun scale-env-vals (env min max newmin newmax)
  (mapcar #'(lambda (pair)
              (list (first pair)
                    (rescale (second pair) min max newmin newmax)))
          env))

(defmethod reverse-envs ((envs simple-note-envs) &key (copy? t))
  (let ((out-envs (if copy?
                      (copy-object envs)
                    envs)))
    (set-number-env out-envs (reverse-env (number-env out-envs)))
    (set-rhythm-env out-envs (reverse-env (rhythm-env out-envs)))
    (set-dur-env out-envs (reverse-env (dur-env out-envs)))
    (set-amp-env out-envs (reverse-env (amp-env out-envs)))
    (set-pitch-env out-envs (reverse-env (pitch-env out-envs)))))

(defmethod invert-envs ((envs simple-note-envs) &key (copy? t))
  ;; nur Tonhoehen umkehren?
  (let ((out-envs (if copy?
                      (copy-object envs)
                    envs)))
    (set-rhythm-env out-envs (invert-env (rhythm-env out-envs)))
    (let ((dur-env (dur-env out-envs)))
      (when dur-env (set-dur-env out-envs (invert-env dur-env))))
    (set-amp-env out-envs (invert-env (amp-env out-envs)))
```

```
    (set-pitch-env out-envs (invert-env (pitch-env out-envs)))))


(defun time-reziproke (time)
  ;; aux fuer reverse-envs
  (if (< time 0)
      1
    (if (> time 1)
        0
      (- 1 time))))

;;;(defun reverse-env (env)
;;;  ;; aux fuer reverse-envs
;;;  (mapcar #'(lambda (pair)
;;;              (cons (time-reziproke (first pair))
;;;                    (rest pair)))
;;;          (reverse env)))

(defmethod variate-envs-with-envs ((envs simple-note-envs) (offset-envs simple-note-envs))
  (let ((variations-list))
    (dotimes (i (number offset-envs))
      (setf variations-list
        (append variations-list
                (list (variate-envs-as-envs-value envs offset-envs i)))))
    (apply #'append-simple-note-envs variations-list)))

;;-----------------------------------------------------------------------------

(defmethod flat-all-envs ((envs-obj simple-note-envs))
  (let ((envs-names
         (remove 'number
                 (mapcar #'SLOT-DEFINITION-NAME
                         (class-slots (class-of envs-obj))))))
    (dolist (env envs-names)
      (set-object envs-obj
                  env (flat (cm::careful-slot-value envs-obj env))))))

(defmethod envs2thread ((envs simple-note-envs) (thread thread) &key (type 'midi-note))
  (let ((unset-notes (make-unset-objects (make-object type) (number envs))))
;;;    (set-container-objects thread unset-notes)
    (remove-all-objects thread)
    (add-objects unset-notes thread)
    (flat-all-envs envs)
    (dotimes (i (object-count thread))
      (set-object-as-envs-value (nth-object i thread) envs))
    thread))

(defmethod thread2envs ((thread thread))
  ;; erstmal fuer  midi-note-thread2simple-note-envs
  (make-object 'simple-note-envs
    :number (object-count thread)
    :rhythm-env (extract-env thread 'rhythm)
    ;; amp-Korrektur bei MIDI-import nicht noetig...
    :amp-env (map-env-values (extract-env thread 'amplitude)
                             (lambda (x)
                               (/ x 127.0)))
    :pitch-env (map-env-values (extract-env thread 'note)
                               #'pitch)))
```

C.6. Die Datei *\in-cm-package.lisp

```
;;;*************************************************************************
;;
;;   Copyright (c) 1999 by Torsten Anders. All rights reserved.
;;   For suggestions, comments and bug reports email to: torsten.anders@hfm.uni-weimar.de
;;
;;   Copyright (c) of Common Music 89-97, 98 Heinrich Taube. All rights reserved.
;;   Copyright (c) of Screamer 1991 Massachusetts Institute of Technology. All rights reserved.
;;                             1992, 1993 University of Pennsylvania. All rights reserved.
;;                             1993 University of Toronto. All rights reserved.
;;
;;   This program is free software; you can redistribute it and/or modify
;;   it under the terms of the GNU General Public License as published by
;;   the Free Software Foundation; either version 2 of the License, or
;;   (at your option) any later version.
;;
;;*************************************************************************

(in-package :cm)

;; aufgerufen von set-container-objects, dies verwendet defcontain
(defun remove-and-unlink-all-objects (container)
  (let* ((bag (slot-value container 'elements))
         (elements (bag-cache bag)))
    ;; variation of cm::remove-all-objects
    (loop for element in elements
        do
          (let ((p (slot-value element 'container)))
            (if (consp p)
                (setf (slot-value element 'container)
                  (delete container p))
              (slot-makunbound element 'container))))
    (setf (bag-cache bag) nil (bag-tail bag) nil (bag-elements bag) nil)
    container))
```

C.7. Die Datei *\def-n-init-arno.lisp

```
;;;*************************************************************************
;;
;;   Copyright (c) 1999 by Torsten Anders. All rights reserved.
;;   For suggestions, comments and bug reports email to: torsten.anders@hfm.uni-weimar.de
;;
;;   Copyright (c) of Common Music 89-97, 98 Heinrich Taube. All rights reserved.
;;   Copyright (c) of Screamer 1991 Massachusetts Institute of Technology. All rights reserved.
;;                             1992, 1993 University of Pennsylvania. All rights reserved.
;;                             1993 University of Toronto. All rights reserved.
;;
;;   This program is free software; you can redistribute it and/or modify
;;   it under the terms of the GNU General Public License as published by
;;   the Free Software Foundation; either version 2 of the License, or
;;   (at your option) any later version.
```

```
;;
;;***************************************************************************

(in-package :cm)

;; the directory arno is installed
(defvar *arno-dir* "/home/t/lisp/cm-etc/screamer-in-cm/arno/arno-nextRevision/")


(defun cm-exports ()
  ;; function by Rick Taube, to export as much as possible of cm, thanks
  (let ((exports '()))
    ;; export everything with an entry in the dictionary...
    (dolist (h *cm-help*)
      (let* ((s (first h))
             (l (length s))
             )
        (when (and (> l 0) (alphanumericp (elt s 0)))
          (push (read-from-string
                 s nil nil
                 :end (or (position #\. s :from-end t) l))
                exports))))
    ;; export stella command functions
    (nconc exports (mapcar #'second *commands*))
    ;; export note names and pitch classes in the standard scale
    (let* ((notes '())
           (fn #'(lambda (k v) v (when (symbolp k) (push k notes)))))
      (maphash fn (scale-tokens *standard-scale*))
      (maphash fn (scale-entries *standard-scale*))
      (nconc exports notes))
    ;; export object classes
    (nconc exports (mapcar #'class-name
                           (class-subclasses (find-class 'timed-object))))
    ;; export item streams
    (nconc exports (mapcar #'class-name
                           (class-subclasses (find-class 'item-stream))))
    exports))

(export
 ;; to avoid conflicts between cl:merge and cm::merge
 (remove 'merge (cm-exports)) 'cm)

;;-----------------------------------------------------------------------------

;; list of file names in the arno dir without extensions
(defvar *arno-files*
    '("object-utils" "arno-kernel" "asa-usw"
      "utils" "in-cm-package" "envelope" "envelope-object"))

(screamer:define-screamer-package :arno
    (:use :cm :clos))

(shadowing-import 'cm::merge 'arno)

(in-package :arno)

(defun add-exension (file-names extension)
  ;; aux: adds given filename extension (as .lisp) to all files of a list of file names
```

```
  (mapcar #'(lambda (name)
              (merge-pathnames cm::*arno-dir*
                               (cm::string-append name extension)))
          file-names))

#|
;; function EXCL:COMPILE-FILE-IF-NEEDED not common,
;; so you have to compile arno completly once by hand
;; (will be replaced by a more userfriendly way later)

;; compile all files of arno
(in-package :arno)
(dolist (path (add-exension cm::*arno-files* ".lisp"))
  (compile-file path))
|#

;; load all files of arno
(dolist (path (add-exension cm::*arno-files* ".fasl"))
  (load path))

(export (append *asa-usw* *arno-kernel* *envelope* *envelope-object* *object-utils* *utils*) 'arno)
;;(shadowing-import '(screamer:fail) 'common-music)
(use-package 'arno 'common-music)

;; import a few screamer functions to cm
(import '(screamer:either screamer:one-value screamer:all-values
          screamer:an-integer-between screamer:a-member-of)
        'common-music)
```

Erklärung

Ich erkläre hiermit, daß ich die Arbeit ohne fremde Hilfe auf Grund der angegebenen Literatur- und Quellennachweise sowie selbst durchgeführter Untersuchungen angefertigt habe.

Weimar, 3. Januar 2000

Literaturverzeichnis

Roman Bartak. on-line guide to constraints programming. `http://kti.ms.mff.cuni.cz/~bartak/constraints/`, 1998.

Roman Barták. Constraint Programming: In Pursuit of the Holy Grail. In *Proceedings of WDS99*, Prague, June 1999.

Ivan Bratko. *PROLOG Programming for Artificial Intelligenz*. Addison-Wesley, Massachusetts, 1986.

Thomas Christaller, Franco di Primo, and Angi Voss, editors. *Die KI-Werkbank Babylon: eine offene und portable Entwicklungsumgebung für Expertensysteme*. Addison-Wesley, 1989.

David Cope. *Computers and Musical Style*. Oxford University Press, 1991.

Digitool Inc. Homepage. `http://www.digitool.com/`.

Kemal Ebcioğlu. An Expert System for Harmonizing Chorales in the Style of J.S. Bach. In Mira Balaban, Kemal Ebcioğlu, and Otto Laske, editors, *Understanding Music with AI: Perspectives on Music Cognition*, pages 295–332. MIT Press, 1992.

Kemal Ebcioğlu. An Expert System for Harmonizing Four-Part Chorales. In Stephan M. Schwanauer and David A. Levitt, editors, *Maschine Models of Music*, pages 384–401. MIT Press, 1993.

John A. Eikenberry. Linux ai/alife mini-howto. `http://www.ai.uga.edu/~jae/ai.html`.

Michael Jampel et al. Constraints FAQ. `http://www.faqs.org/faqs/constraints-faq/`, 1996.

Franz Inc. Homepage. `http://www.franz.com/`.

Paul Graham. *On Lisp, Advanced Techniques for Common Lisp*. Prentice Hall, 1994.

Paul Graham. *ANSI Common Lisp*. Prentice Hall, 1995. deutsch 1999.

Lejaren Hiller and Leonard Isaacson. Musical Composition with a High-Speed Digital Computer. In Stephan M. Schwanauer and David A. Levitt, editors, *Maschine Models of Music*, pages 8–21. MIT Press, 1993. Copyright 1958, *Journal of the Audio Engineerins Society*.

HyperSpec. Common Lisp HyperSpec. `http://www.harlequin.com/`, 1996.

LOOM User Guide, Version 1.4. ISX Corporation, August 1991.

Loom Tutorial, Version 2.1. ISX Corporation, Mai 1995.

Michael Jampel, David Joslin, and Peg Eaton. Constraints archive. `http://www.cs.unh.edu/ccc/archive/`, 1996.

Knud Jeppesen. *Kontrapunkt*. Breitkopf & Härtel, Leipzig, 1930. 4. deutsche Auflage, 1971.

Sonya E. Keene. *Object-Oriented Programming in COMMON LISP*. Addison-Wesley and Symbolics Press, 1989.

Grzegorz Kondrak. A Theoretical Evaluation of Selected Backtracking Algorithms. Master's thesis, University of Alberta, 1994.

Vipin Kumar. Algorithms for constraints satisfication problems: A survey. *AI Magazine*, 1992.

Tobias Kunze. Common Music: Kompositionssprache und -Umgebung in Common Lisp. Ein Überblick. *Mitteilungen der Deutschen Gesellschaft für Elektroakustische Musik*, Bände 13-15, 1994.

Mikael Laurson. *PatchWork, PWConstraints*. IRCAM, Paris, first english edition, Oktober 1996. documentation.

David A. Levitt. A Representation for Musical Dialects. In Stephan M. Schwanauer and David A. Levitt, editors, *Maschine Models of Music*, pages 452–469. MIT Press, 1993.

Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

Dieter de la Motte. *Kontrapunkt*. Bärenreiter-Verlag, 1981.

Dick Pountain. Constraints logic programming. *BYTE*, 1995.

Curtis Roads. *The Computer Music Tutorial*. MIT Press, 1996.

Camilo Rueda and Antoine Bonnet. *PatchWork Situation, Library for musical constraints programming*. IRCAM, Paris, first english edition, 1996.

Arnold Schoenberg. *Vorschule des Kontrapunkts*. Universal Edition, Wien, 1963. deutsch 1977.

Arnold Schönberg. *Harmonielehre*. Universal Edition, Wien, 1911. 7. Auflage 1986.

Jeffrey Mark Siskind. *Screaming Yellow Zonkers*. MIT Artificial Intelligence Laboratory, 1991.

Jeffrey Mark Siskind and David Allen McAllester. Screamer: A portable efficient implementation of nondeterministic common lisp. Technical report, University of Pennsylvania Insitute for Research in Cognitive Science, 1993.

Guy L. Steele. *Common Lisp the Language, 2nd Edition*. Digital Press, 1990.

Heinrich Taube. Common Music. `http://ccrma-www.stanford.edu/CCRMA/Software/cm/cm.html`, a.

Heinrich Taube. Common Music Dictionary. `http://ccrma-www.stanford.edu/CCRMA/Software/cm/dict/index.html`, b.

Heinrich Taube. Stella Tutorial. `http://ccrma-www.stanford.edu/CCRMA/Software/cm/tutorials/stella/toc.ht\%ml`, c.

Heinrich Taube. Stella: Persistent Score Editing in Common Music. *Computer Music Journal*, 17:4, 1994.

Heinrich Taube and Tobias Kunze. Capella: A graphical interface for algorithmic composition. In *Proceedings of the 1995 International Computer Music Conference*, pages 377–380. ICMA, 1995.

Patrick Henry Winston. *Artificial Intelligence*. Addison-Wesley, 1984.

Patrik Henry Winston and Bertold Klaus Paul Horn. *LISP*. Addison-Wesley, 1993 3rd edition, 1989.