Brandenburgische Technische Universität Cottbus

Fakultät 1: Mathematik, Naturwissenschaften und Informatik

Bachelorarbeit

Testen mit Musik

eingereicht von Denny Bayer

Matrikel: 2511181

Studiengang Informations- und Medientechnik

Abgabe: Cottbus, August 2008

Betreuer: Prof. Dr.-Ing. Monika Heiner

Dipl.-Ing. (FH) Ronny Richter

Aufgabenstellung

Eine Anwendung von Petri-Netzen ist die Modellierung von Software. Aufgabe ist die Realisierung einer Möglichkeit, Software zum Zwecke des Tests hörbar zu machen. Mit dem am Lehrstuhl Datenstrukturen und Softwarezuverlässigkeit entwickelten Editor *Snoopy* können Petri-Netze animiert werden. Nun soll der Editor um einen Musik-Netz-Typ erweitert werden, mit dem die Petri-Netz-basierte Simulation der Software hörbar wird.

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Denny Bayer

Inhalt

1 Einleitung

1.1 Motivation

Die Arbeit am Computer findet heute überwiegend mit Hilfe grafischer Benutzeroberflächen statt. Seit ihrer Einführung in den 1980er Jahren wurde großer Aufwand betrieben um Oberflächen zu entwickeln, die einerseits intuitiv bedienbar sind und gleichzeitig viele Information darstellen können. Vielfach finden Metaphern Anwendung, wie etwa die des Papierkorbs, des Schreibtisches oder Ordners. Auch wird durch die Darstellung von Anwendungen in Fenstern ermöglicht, diese übereinander zu schichten. Das hat allerdings den Nachteil, dass Informationen verdeckt sein können.

Während das visuelle Medium breite Anwendung findet, wird – von Spielen und Systemen für sehbehinderte Menschen abgesehen – erstaunlich wenig Audio verwendet. Dabei bietet Audio einige Vorteile. Zum Beispiel kann man bei dem oben genannten Problem der Verdeckung von Informationen in anderen Fenstern auf Veränderungen mit Hilfe von Klängen hinweisen. Desweiteren eignet sich Audio besonders um zeitliche Informationen darzustellen, während sich räumliche Information eher visualisieren lassen. Möchte man z.B. während in einem Fenster der Fortschritt eines Kompiliervorgangs angezeigt wird, in einem anderen Fenster einen Text verfassen, ist es hilfreich, wenn man akustisch darauf hingewiesen wird, wenn der Kompiliervorgang beendet ist. Damit kann man sich in der Zwischenzeit ganz auf die aktuelle Aufgabe, das Schreiben, konzentrieren [Bre98].

Trotz der Vorteile wird Audio spärlich verwendet. Das mag zum einen daran liegen, dass günstige Ausgabegeräte relativ jung sind, während ihre visuellen Gegenspieler schon lange zur Standardausrüstung gehören. Dem entsprechend weit ist auch ihre Entwicklung, bzw. die der Anwendungen dafür fortgeschritten. Wichtig ist auch zu beachten, dass Audio von Natur aus sehr aufdringlich ist. Wenn eine Anwendung Audio benutzt, ist es sehr schwer dies zu ignorieren. Das kann besonders problematisch sein, wenn sich mehrere Personen einen Arbeitsraum teilen. Man kann hier argumentieren, dass das Problem durch die Benutzung von Kopfhörern einfach gelöst werden kann, aber das führt unweigerlich zu einer Einschränkung der Kommunikation zwischen den Personen, und damit zu neuen Problemen.

Ein wichtiger Faktor, der für die Verwendung von Audio spricht, ist allerdings die menschliche Hörwahrnehmung selbst. So sind geschulte Hörer in der Lage sich ganze Sinfonien zu merken, subtile Unterschiede zwischen verschiedenen Aufführungen festzustellen und mit Sicherheit zu erkennen, wenn jemand einen falschen Ton spielt. Bilder kann sich der Mensch nicht mit einer solchen Genauigkeit merken[Vic03]. Zwar kann

der Mensch sehr viele Farben unterscheiden, jedoch fällt es ihm schwer sich Bilddetails so genau zu merken wie Gehörtes. Hier gibt es also ein großes Potential für Audio.

Allerdings sind die Unterschiede in diesen Fähigkeiten zwischen geschulten und nicht geschulten Hörern groß. So sind Menschen mit absolutem Gehör in der Lage die Höhe eines gehörten Tones genau zu benennen (passiv), bzw. einen gewünschten Ton einfach anzusingen (aktiv). Diese Fähigkeit ist jedoch sehr selten. Nur einer von 10000 Menschen hat ein absolutes Gehör[Sac07]. Wesentlich weiter verbreitet ist das sogenannte relative Gehör. Solche Menschen können zwar keine absolute Tonhöhe erkennen, wohl aber die Intervalle zwischen Tönen.

Trotz dieses Potentials gibt es einige Probleme denen man bei der Nutzung von Musik in Benutzerschnittstellen gegenübersteht. Zusammengefasst sind dies [Alt95]:

- Obwohl Musik komplexe Informationen tragen kann, sind diese emotionaler Natur und deswegen mitunter ungeeignet für Interface-Design,
- Musik ist stark an regionale Kulturen gebunden, sodass ihre Verwendung nur begrenzt Anklang finden wird,
- Quantitative Daten können nicht durch Klänge oder Musik ausgedrückt werden,
- Audio ist ein aufdringliches Medium und daher in Situationen ungeeignet, in denen sich mehrere Person einen Arbeitsraum teilen,
- Nur geschulte Hörer können musikalisch dargestellte Daten einschätzen und verstehen.

Es geht aber auch gar nicht darum, dass Programme Musik machen. Vielmehr möchte man Informationen auditiv darstellen, z.B. um den visuellen Kanal zu entlasten. Es gilt also, sich der genannten Schwächen von Audio bewusst zu sein und innerhalb bestimmter Grenzen zu bleiben, wenn man Anwendungen dafür entwickelt:

Musik ist stark an die regionale Kultur gebunden, jedoch ist die westliche Musik mit ihrer wohltemperierten Stimmung mittlerweile weltweit verbreitet, sodass hier am wenigsten Probleme zu erwarten sind [Alt95]. Weiterhin gilt es, Informationen nicht zu komplex zu kodieren, sie z.B. nicht ausschließlich auf die Tonhöhe abzubilden, damit sie unterscheidbar bleiben [Vic99]. Auf diesem Gebiet wurden bereits Untersuchungen angestellt, auf die im weiteren Verlauf dieser Arbeit noch eingegangen wird.

Doch lange bevor überhaupt darüber nachgedacht wurde, Klänge in Programme einzubauen, nahmen in den frühen Tagen des Computers bereits Programmierer Audio zu Hilfe. Eine viel zitierte Geschichte ist die von Programmierern, die ein Radio nutzten, um die von ihren Computern produzierten Interferenzen einzufangen und damit CPU-Aktivitäten zu überprüfen und sogar fehlerhaftes Programmverhalten zu erkennen [Vic03]. Das ist ein ganz anderer Ansatz, der aber auch den Nutzen von Audio aufzeigt. Anhand der Geräusche aus dem Radio konnten sie feststellen, ob ihr Programm korrekt

ausgeführt wurde, oder ob es sich beispielweise in einem Deadlock oder einer Endlosschleife oder ähnlichem befand.

Mittlerweile wächst das Interesse an Audio in der Mensch-Maschine-Kommunikation, denn es hat in den letzten Jahren vermehrt Forschungsarbeit auf diesem Gebiet stattgefunden Dies liegt daran, dass durch die hohe Spezialisierung der grafischen Benutzeroberflächen sehbehinderte Menschen stark benachteiligt sind, und auch weil Audio die Mensch-Maschine-Kommunikation verbessern kann, wenn es richtig eingesetzt wird. Diese Arbeit will einen Beitrag dazu leisten, durch die Implementierung von akustischen Ausgaben in ein Werkzeug zur Modellierung von Petri-Netzen.

1.2 Gliederung der Arbeit

Das Ziel dieser Arbeit ist es, eine Möglichkeit zu erarbeiten, durch Petri-Netze dargestellte Software in geeigneter Form mit Audio anzureichern, um so ein besseres Verständnis dafür zu erhalten. Es wird aber auch versucht, einen Ansatz zu finden, der ein möglichst breites Anwendungsfeld ermöglicht, das über Software hinausgeht und für Petri-Netze im Allgemeinen sinnvoll ist.

Dafür werden zunächst Petri-Netze vorgestellt und definiert. Dann werden einige Beispiele für bisherige Untersuchungen zur Verwendung von Audio in Software betrachtet. Die Beispiele geben einen kurzen Überblick darüber, mit welchen verschiedenen Zielen Audio verwendet wird. Diese sind z.B. Verbesserung der Benutzerfreundlichkeit, Debugging von Programmen oder Barrierefreiheit. Diese Betrachtungen dienen aber auch dazu, die in den Beispielen verwendeten Ansätze und Ergebnisse auf Verwendbarkeit für diese Arbeit zu prüfen, bzw. sie eventuell weiterzuentwickeln. Daraus wird dann ein Musik-Netz-Typ entwickelt werden – ein Petri-Netz, das um Musikobjekte erweitert wird, die dann bei der Animation zu Gehör kommen.

Weiterhin wird untersucht, welche Anforderungen an das Tonmaterial gestellt werden. Daraus wird ein Konzept für dessen Aufbau erstellt. Im Anschluss daran wird die zur Implementierung verwendete Applikation *Snoopy* vorgestellt und beschrieben, welche Änderungen an der Software vorgenommen wurden. Dabei ist es besonders wichtig, die neuen Funktionen in Einklang zu bringen, also etwa die bisherigen Konzepte der Nutzerinteraktion weiterzuverwenden.

Neben der Implementierung besteht eine weitere Aufgabe darin, das entworfene Tonmaterial umzusetzen, das heißt also Audio-Dateien zu erstellen, die mit *Snoopy* genutzt werden können. Dafür wird eine Reihe von Variationen erstellt, um zumindest kleinere Graphen mit genügend verschiedenen Klängen versehen zu können. Die Implementierung soll dabei so gewählt werden, dass der Nutzer auch selbst Klänge hinzufügen kann.

Schließlich wird das Ergebnis der Arbeit getestet, beurteilt und auf nötige oder sinnvolle Erweiterungen geprüft, die in weiteren Arbeiten umsetzbar wären.

2 Vorbetrachtungen

2.1 Petri-Netze

Petri-Netze wurden 1962 von Carl Adam Petri als ein mathematisches Modell eingeführt, mit dem nebenläufige Systeme beschrieben werden können.

Nachfolgend werden Petri-Netze und einige Begriffe, die mit ihnen in Zusammenhang stehen, vorgestellt.

2.1.1 Definition

Ein Petri-Netz ist ein gerichteter, bipartiter Graph. Seine Knoten heißen Plätze (auch Stellen) und Transitionen. Sie sind durch gerichtete Kanten miteinander verbunden. Grafisch werden Plätze durch Kreise und Transitionen durch Rechtecke dargestellt. Zwischen gleichartigen Knoten gibt es keine Kanten (da bipartit). Jeder Kante des Netzes ist auch ein Gewicht zugeordnet, das an der Kante notiert wird. Ein Gewicht von *1* wird in diesem Fall oft weggelassen. Das Gewicht bezeichnet auch die Anzahl gleichgerichteter Kanten zwischen den Knoten. Anstatt also zwei gleiche Kanten (mit dem Gewicht *1*) zu zeichnen, kann man auch eine Kante mit dem Gewicht *2* zeichnen.

Führt eine Kante von einem Platz zu einer Transition hin, heißt dieser Platz Vorplatz der Transition. Im umgekehrten Fall heißt er Nachplatz. Die Mengen der Vor- und Nachplätze einer Transition werden auch Vor- bzw. Nachbereich genannt. Transitionen können schalten. Dabei kommt ein weiteres Netzelement ins Spiel: Die Marken.

Marken befinden sich auf Plätzen. Beim Schaltvorgang werden von den Plätzen des Vorbereichs entsprechend des jeweiligen Kantengewichts Marken entfernt und auf den Plätzen des Nachbereichs den Kantengewichten entsprechend viele Marken erzeugt. Damit eine Transition schalten kann, müssen sich also auf allen Vorplätzen genügend Marken befinden. Ist die Bedingung erfüllt, ist die Transition schaltbereit. Eine schaltbereite Transition heißt auch *aktiviert* oder *hat Konzession*. Aktivierte Transitionen können zu einem beliebigen Zeitpunkt schalten. Sind mehrere Transitionen zur gleichen Zeit aktiviert, kann irgendeine von ihnen schalten. Die Ausführung von Petri-Netzen ist also nichtdeterministisch.

Zwischen aktivierten Transitionen können Konflikte auftreten. Das passiert immer dann, wenn zwei Transitionen die gleiche Marke eines Vorplatzes benötigen, um zu schalten. In diesem Fall müssen Regeln zur Lösung des Konfliktes definiert werden. Es kann z.B. eine der Transitionen zufällig gewählt werden.

2.1.2 Formale Beschreibung

Ein Petri-Netz kann beschrieben werden als Tupel $N = (P, T, F, W, M_0)$ mit

i. $P = \{p_1, p_2, p_3, \dots\ p_m\}$ als Menge der Plätze mit $m \in \mathbb{N}$
ii. $T = \{t_1, t_2, t_3, \dots\ t_n\}$ als Menge der Transitionen mit $n \in \mathbb{N}$
iii. $P \cap T = \emptyset,\ P \cup T \neq \emptyset$
iv. $F \subseteq (P \times T) \cup (T \times P)$ als Menge der Kanten
v. $W: F \to \mathbb{N}$ als Gewichtung bzw. Kosten der Kanten
vi. M_0 als Anfangsmarkierung

Folgende Teilmengen von T sind definiert:

$\bullet\, t = \{p \in P | (p, t) \in F\}$ ist der Vorbereich von t
$t \bullet = \{p \in P | (t, p) \in F\}$ ist der Nachbereich von t

Eine Transition heißt aktiviert, wenn gilt:

$$\forall p \in \bullet\, t: M(p) \geq W(p, t)$$

2.1.3 Schaltvorgang

Eine aktivierte Transition t kann schalten, indem sie die Markierungen aller Vorplätze p um $W(p, t)$ reduziert und auf den Nachplätzen p' $W(t, p')$ Marken erzeugt. Für die so entstehenden Folgemarkierungen M' gilt:

$$\forall p \in P: M'(p) = M(p) - W(p, t) + W(t, p)$$

Durch das Verhältnis der Markierungen M und M' lassen sich die Transitionen in *markenerhaltende, markenproduzierende* und *markenkonsumierende* Transitionen einteilen.

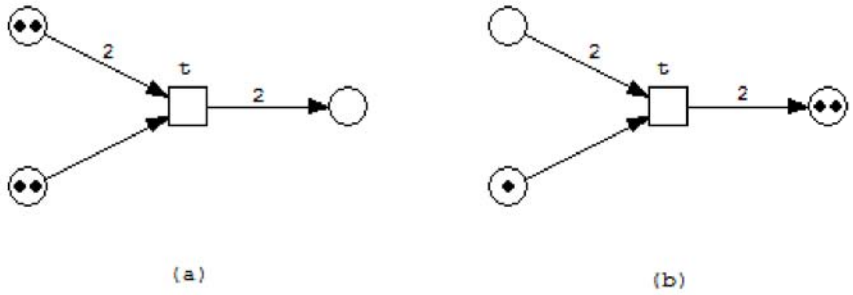

Abbildung 2.1: Schalten einer Transition: (a) Markierung vor dem Schalten von t; (b) Markierung nach dem Schaltvorgang von t

2.1.4 Lebendigkeit

Eine Transition t heißt lebendig, wenn t, ausgehend von der Anfangsmarkierung M_0, durch jede Folgemarkierung aktivierbar ist.

Eine Markierung M heißt lebendig, wenn unter ihr alle Transitionen lebendig sind.

Ein Petri-Netz heißt lebendig, wenn alle seine Transitionen lebendig sind.

2.1.5 Erweiterungen

Für einige Anwendungsgebiete ist es nötig bzw. hilfreich, konventionelle Petri-Netze um spezifische Eigenschaften zu erweitern. Hier gibt es die zeiterweiterten Petri-Netze, deren Transitionen beim Schalten Zeit verbrauchen. Ein weiteres Beispiel sind die farbigen Petri-Netze. Bei den ursprünglichen Petri-Netzen sind die Marken nicht unterscheidbar. Dies wird hier durch verschieden farbige Marken ermöglicht.

Neben weiteren Typen gibt es z.B. Netze mit Hierarchien, in denen Teilnetze als einzelne Knoten dargestellt werden können, wodurch sie beliebig detailierte bzw. abstrahierte Sichten ermöglichen, oder priorisierte Petri-Netze, in denen die Transitionen mit Prioritäten versehen sind, sodass unter mehreren aktivierten Transitionen diejenige mit der höchsten Priorität schalten darf.

2.1.6 Anwendung

Petri-Netze werden zur Modellierung verschiedenster Systeme verwendet. Es lassen sich Beziehungen zwischen Bedingungen und Ereignissen darstellen, indem man Bedingungen durch Plätze modelliert, wobei die gewünschte Markierung eines Platzes die Erfüllung einer Bedingung darstellt, und Ereignisse mit Hilfe von Transitionen, bei denen das Eintreten des jeweiligen Ereignis, dem Schalten der Transition gleichkommt.

Petri-Netze werden z.B. zur Modellierung biochemischer Netzwerke, Ablaufsteuerungen oder zur Verifizierung von Softwaresystemen genutzt. Abbildung 2.2 zeigt ein simples Beispiel einer IF-Anweisung, die als Petri-Netz dargestellt wurde. Die Modellierung muss aber nicht unbedingt auf der Ebene von Programmiersprachkonstrukten stattfinden. Sie kann je nach Anwendung abstrahiert werden, z.B. wenn man Prozesswechsel oder den Umgang mit Betriebsmitteln darstellen möchte.

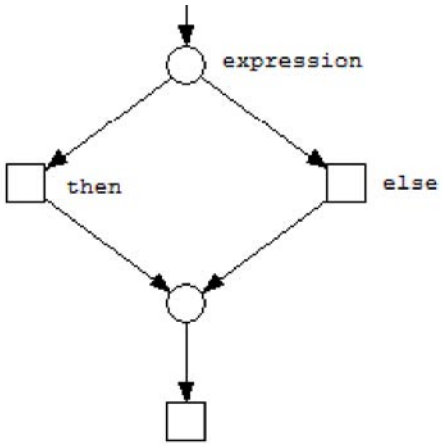

Abbildung 2.2: IF-ELSE-Konstrukt als Petri-Netz

2.2 Bisherige Arbeiten

Nachfolgend werden einige Arbeiten vorgestellt, welche die Verwendung von Audio in Softwaresystemen behandeln.

2.2.1 Soundtrack

Schon Ende der achtziger Jahre entwickelte Edwards [Edw95] das *Soundtrack*-System, ein Textverarbeitungsprogramm für Blinde mit einer Maus-gesteuerten Benutzerschnittstelle. Dabei war der Bildschirm in acht Fenster bzw. Menüs eingeteilt, die, wenn man die Maus darüber bewegt, bestimmte Klänge (die sich in der Tonhöhe unterschieden) abgespielten. Beim Klicken in eines dieser Fenster wurde ihr Name wiedergegeben. Durch Doppelklicken wurde das Fenster seinerseits in mehrere Bereiche eingeteilt, an die verschiedene Aktionen und ebenfalls Klänge geknüpft waren. Per Doppelklick konnten diese Aktionen ausgeführt werden.

Dieses System wurde ausschließlich für Forschungszwecke entwickelt. Obwohl es bewies, dass es möglich ist, eine Maus-basierte Umgebung für sehbehinderte Menschen zu schaffen, hatte das System auch einige Schwächen. So wurde angeführt, dass ein Nutzer sich einfach mehr merken muss, als bei einem normalen System [Edw95]. Während man z.B. sonst nur wissen muss, dass der Befehl zum Ausschneiden von Text *Strg-X* ist, muss der Soundtrack-Anwender wissen, dass der Befehl im Editier-Menü ist, wo sich das Editier-Menü befindet und wo der Befehl innerhalb des Menüs ist.

2.2.2 Earcons

Nicht an sehbehinderte, sondern normal sehende Menschen richtet sich das Konzept der *Earcons*. Ihr Name leitet sich von den sogenannten *Icons* ab – grafische Symbole, die Programme, Dateien oder ähnliches repräsentieren, auf die der Benutzer klicken kann. Brewster [Bre98] erforschte ihre Bedeutung bei der Verwendung mit Elementen grafischer Benutzeroberflächen. Es handelt sich dabei um spezielle Klänge, die z.B. an Interaktionselemente, Aktionen oder Hinweismeldungen geknüpft werden und so vom Benutzer damit assoziiert werden können. Dadurch soll der Nutzer eine Rückkopplung von der Software bekommen, die allein durch grafische Elemente nicht gegeben ist.

Will ein Nutzer beispielsweise auf einen Button klicken und bewegt dabei versehentlich die Maus, bevor er die Maustaste loslässt (also vom Button abrutscht), wird die gewünschte Aktion nicht ausgeführt. Dies ist möglicherweise für den Nutzer nicht sofort ersichtlich. Das Problem kann durch ein Earcon gelöst werden, indem ein entsprechender Klang abgespielt wird, wenn die Aktion ausgeführt wird, bzw. keiner, wenn der Fehler auftritt. Die verwendeten Klänge sind dabei aber nicht willkürlich, denn laut Brewster sieht das Konzept vor, dass sie hierarchisch aufgebaut sind.

Abbildung 2.3 zeigt das Hierarchiekonzept am Beispiel von Fehlermeldungen. So wird eine Fehlermeldung durch einen Ton mit einer bestimmten Höhe dargestellt. Das Instrument gibt die Art des Fehlers an. Eine zusätzlich zu dem Grundton erklingende Melodie wird genutzt, um den Fehler noch weiter zu spezifizieren. Damit die verwendeten Klänge jedoch nützlich sind, ist es wichtig, dass sie zumindest innerhalb einer Anwendung eindeutig zugeordnet werden können und nicht in verschieden Kontexten unterschiedliche Bedeutungen haben. Brewster schlägt deshalb sogar systemweit eindeutige Klänge vor, die ähnlich den sogenannten *Desktop Themes* bei Farben und Schriftarten, vom Benutzer global festgelegt werden können.

In Untersuchungen fand man heraus, dass die Verwendung von Earcons z.B. bei Buttons, Menüs, Scrollbalken, oder Fehlermeldungen tatsächlich hilft, auftretende Fehler, wie das oben genannte Abrutschen von Buttons zu vermeiden. Im Falle von Hinweismeldungen wird außerdem erreicht, dass der Benutzer seine aktuelle Tätigkeit (z.B. das Schreiben eines Briefs) nicht unterbrechen muss, um die Information zu erhalten (z.B. dass ein Programm fertig kompiliert ist). Er kann seine Augen, bzw. seine Aufmerksamkeit also weiterhin auf den Brief richten, während er durch eine grafische Mitteilung gestört werden würde [Bre98].

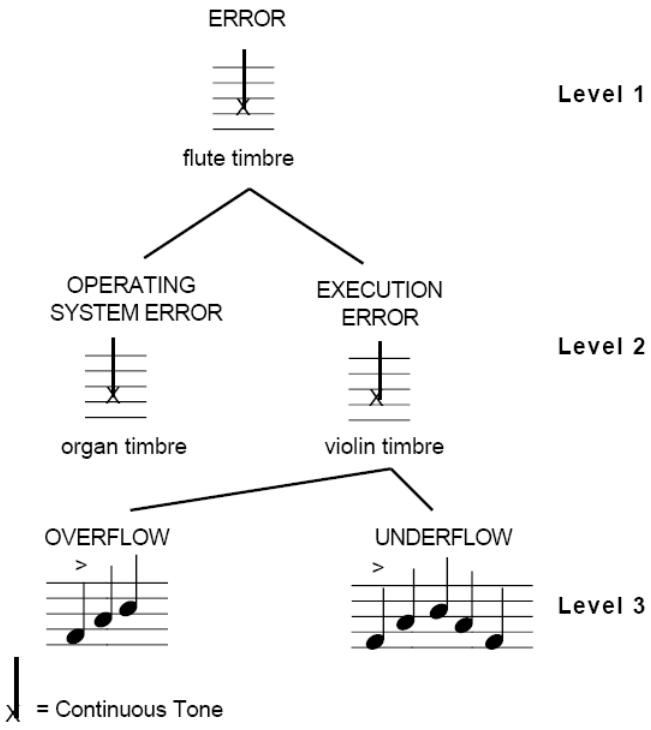

Abbildung 2.3: Hierarchie von Klängen [Bre98]

2.2.3 CAITLIN

CAITLIN [Vic99] [Vic03] (Computer Audio Interface to Locate Incorrent Nonsense) ist ein von dem Briten Paul Vickers entwickeltes System. Dabei handelt es sich im Prinzip um einen Präprozessor für *PASCAL*-Programme, mit dem der Code um Audioausgaben erweitert wird, die dann beim Ausführen des Programmes abgespielt werden. Das System richtet sich speziell an Programmieranfänger, die dadurch ihre Programme besser verstehen und Fehler finden sollen. Vickers führte Tests mit Probanden durch, anhand derer er belegen konnte, dass durch Audio tatsächlich ihr Verständnis für die Programme verbessert werden konnte. Auch fanden die Probanden in den ihnen vorgelegten fehlerhaften Programmen mehr Fehler mit Hilfe der Audiounterstützung als ohne.

Vertont werden von *CAITLIN* Programmiersprachkonstrukte wie Schleifen, *IF*- und *CASE*-Anweisungen. Dabei wird aber nicht nur das bloße Ausführen solcher Konstrukte hörbar gemacht. Vielmehr werden z.B. bei einer *FOR*-Schleife der Einstieg, die Iterationsschritte und der Ausstieg, durch einen Klang dargestellt. Die Klänge sind dabei, ähnlich dem Earcon-Konzept, hierarchisch strukturiert (Abbildung 2.4).

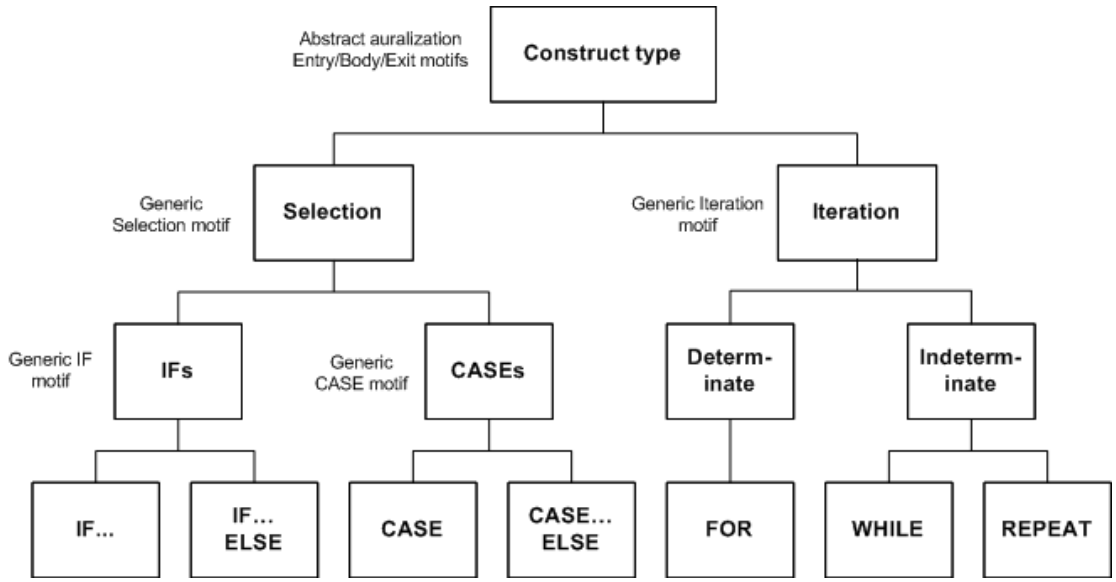

Abbildung 2.4: Hierarchischer Aufbau der Klänge von CAITLIN [Vic99]

Jedes Konstrukt bekommt ein kurzes musikalisches Thema zugeordnet an dem man es erkennen kann. Zudem soll es auch seinen Charakter widerspiegeln, ähnlich den Themen die Sergej Prokoviev den Charakteren in *Peter und der Wolf* (1936) gegeben hat [Vic03]. Der Aufbau eines solchen Themas soll im Folgenden am Beispiel einer als wahr ausgewerteten IF-Anweisung erläutert werden (Abbildung 2.6).

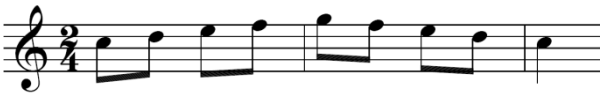

Abbildung 2.5: Generisches Anweisungsmotiv [Vic08]

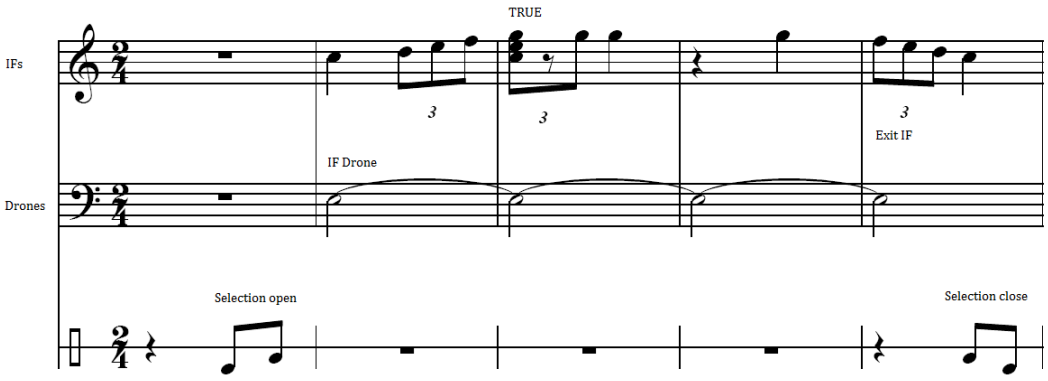

Abbildung 2.6: Positiv ausgewertetes IF-Konstrukt [Vic03]

Der Einstieg in das Konstrukt wird durch zwei Schläge auf eine Kuhglocke repräsentiert. Dies entspricht der obersten Ebene in der Hierarchie und gibt noch keine Information darüber, um welche Art von Konstrukt es sich handelt. Während der gesamten Ausführung der *IF*-Anweisung, in der sich beliebig viele Anweisungen befinden können, wird ein Orgelton ausgehalten (mittleres Notensystem). Sind mehrere Anweisungen verschachtelt, erklingen hier mehrere unterschiedlich hohe Töne. Die unteren beiden Notensysteme entsprechen also der obersten Hierarchieebene aus Abbildung 2.4.

Das oberste Notensystem zeigt das *IF*-Motiv, welches eine Variation des generischen Anweisungsmotivs (Abbildung 2.5) darstellt. Die verschiedenen Anweisungs- und Iterationsmotive unterscheiden sich neben der konkreten Melodie noch im Timbre. Die Auswertung auf *TRUE* wird durch den Dur-Akkord in Takt 3 verdeutlicht. Bei einem *FALSE* wäre es Moll, - was Vickers damit begründet, dass selbst Nicht-Musiker den Unterschied gut wahrnehmen können. Wenn das Konstrukt abgearbeitet ist, wird die Melodie vom Anfang invertiert gespielt, um ihr einen schlüssigen Charakter zu geben. Auf der ersten Ebene klingt der Orgelton aus und die Kuhglocke erklingt noch einmal. Es ist also auch möglich, einfach auf der obersten Ebene die Dauer des Konstrukts zu verfolgen.

2.2.4 Mailänder Konzept

Dieses Konzept [Lev95] befasst sich mit computergestütztem Modellieren von Musikstücken mit Hilfe von Petri-Netzen und wurde Ende der 1980er Jahre an der Universität Mailand entwickelt. Das Komponieren nach diesem Konzept erfolgt in zwei Arbeitsphasen: Dem Erzeugen spezieller Petri-Netze und dem Ausführen dieser Netze. Die Netze sind dabei um bestimmte Eigenschaften erweiterte Stellen-/Transitions-Netze. Melodien (hier Musikobjekte genannt) werden an Stellen geknüpft und Algorithmen an Transitionen. Es stehen aber nicht alle Musikobjekte schon zu Beginn fest. Vielmehr werden, ausgehend von einem oder mehreren Musikobjekten, mit Hilfe der Algorithmen Transformationen auf ihnen ausgeführt, durch die dann weitere Musikobjekte entstehen. Schaltet eine Transition, wird das Musikobjekt des Vorbereichs durch den Algorithmus verändert und an die Stelle des Nachbereichs kopiert.

Als Algorithmen werden z.B. Verfahren wie Spiegelung (an einem bestimmten Ton), Transposition, Invertierung oder Rotation verwendet. Das Konzept sieht vor, dass das während der Ausführung entstehende Musikstück in eine Datei kopiert wird.

Da Musik immer auch mit einem Metrum verbunden ist, wird festgelegt, dass immer, wenn eine Transition schalten kann, sie es ohne Zeitverzögerung tun muss, d.h. es gilt die Sofortschaltregel. Sobald eine Stelle mit einer Marke belegt ist, wird das entstandene Musikobjekt der Stelle zugeordnet und in die Datei kopiert. Erst danach ist die Transition im Nachbereich aktiviert. Der Schaltvorgang einer Transition dauert so lange, wie der Algorithmus zur Ausführung benötigt. Bei Konflikten wird eine zufällige Transition gewählt. Abbildung 2.7 zeigt ein einfaches Netz, in dem das Musikobjekt A durch die gegebenen Algorithmen transformiert wird. Ergebnis ist ein Musikstück, gebildet aus A, A‘ und A‘‘.

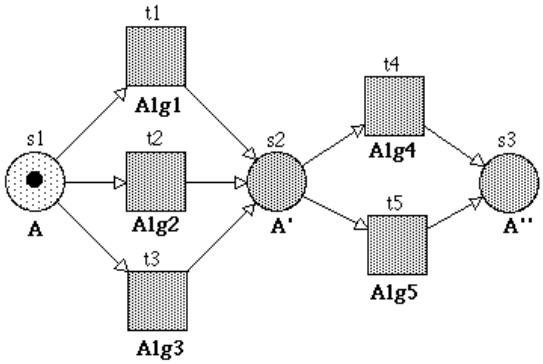

Abbildung 2.7: Einfaches Musik-Netz [Lev95]

3 Entwurf

Audio kann in vielen Gebieten Anwendung finden. Es kann genutzt werden, um sehbehinderten Menschen die Nutzung von Computern zu ermöglichen, oder auch für Normalsehende, zur Verbesserung der Benutzerfreundlichkeit. Es kann helfen, Programme besser zu verstehen und sogar Fehler leichter zu finden. Die obigen Beispiele zeigen, dass die Verwendung von Audio tatsächlich eine Bereicherung sein kann. Im Folgenden soll betrachtet werden, inwiefern sich das Konzept des *CAITLIN*-Systems für die Anwendung auf Petri-Netze eignet und ob das Mailänder Konzept auch zum Testen benutzt werden kann.

3.1 Entwurf des Musik-Netz-Typs

Bei dem Mailänder Konzept (MK) enthalten Plätze Musikobjekte (Melodien), die durch Schaltvorgänge von Transitionen modifiziert werden. Das dabei entstehende Musikstück spiegelt damit die Animation mit allen Schaltvorgängen wider. Man erhält die akustische Darstellung der Animation aber erst im Nachhinein. Beim MK steht allerdings die Musik im Vordergrund. Das Petri-Netz ist dabei der Weg zum Ziel, was in dieser Arbeit nicht der Fall ist. Vielmehr ist die Musik ein Hilfsmittel für die Arbeit mit dem Petri-Netz.

Das MK in seiner Reinform ist für das Ziel dieser Arbeit aus einer Reihe von Gründen ungeeignet. Zwar erhält man eine eindeutige, akustische Entsprechung, aber man muss damit rückwärts auf die Ausführung des Netzes schließen, was sehr aufwändig scheint. Eine sofortige, akustische Rückkopplung der Software während der Animation, wäre hier sinnvoller. Dadurch, dass die meisten Musikobjekte erst während der Animation entstehen, müsste man die Algorithmen zur Auswertung nochmal selbst nachvollziehen. Ein weiteres Problem liegt in den Algorithmen selbst. Um ein musikalisch qualitatives Ergebnis zu erhalten, dürfte man die Algorithmen nicht wahllos im Netz vergeben, da man durch unbedachtes Anwenden von Transpositionen oder Spiegelungen die Tonalität verlassen kann, was für die meisten Nutzer sicher eher störend ist.

Für diese Arbeit entsteht aber auch kein größerer Nutzen, wenn das Tonmaterial erst während der Animation erzeugt wird; d.h. es würde genügen, wenn das Tonmaterial schon vor der Animation feststeht (wie auch beim MK in den Anfangsstellen). Anstatt also das den Plätzen zugeordnete Tonmaterial bei Schaltvorgängen zu modifizieren, kann man es einfach den Transitionen zuordnen und bei ihren Schaltvorgängen abspielen, und erhält so eine direkte akustische Informationen über die Aktivität des Systems. Voraussetzung ist, dass die Klänge der Transitionen eindeutig sind.

Adaptiert man die Funktion des *CAITLIN*-Systems für Petri-Netze, erhält man das gleiche Ergebnis. Das *CAITLIN*-System stellt Systemaktivitäten, dort ist es die Ausführung von Programmcode bzw. bestimmten Sprachkonstrukten, durch Audio dar. Den verschiedenartigen Aktivitäten sind dabei eindeutige Klänge zugeordnet, die bei Ausführung zu Gehör kommen. Auf Petri-Netze lässt sich dies dahingehend anwenden, dass man den Transitionen einen bestimmten Klang zuordnet, der bei Aktivität, also ihrem Schalten abgespielt wird. Dieser Ansatz soll nun umgesetzt werden.

3.2 Entwurf des Tonmaterials

Mit dem jetzigen Entwurf des Musik-Netzes besteht die Information, die ein Klang trägt darin, das Schalten seiner Transitionen zu verdeutlichen bzw. die schaltende Transition eindeutig zu identifizieren. Generell ist es sicherlich richtig, die Klänge so simpel wie möglich zu gestalten, damit sie leicht verständlich sind. Allerdings müssen sie auch komplex genug sein, damit man eine genügend große Anzahl unterscheidbarer Klänge erstellen kann, denn es soll möglich sein, jede Transition eines Netzes mit einem eigenen Klang zu versehen. Das Tonmaterial selbst wird als Teil dieser Arbeit erstellt.

Es gibt nun einige Möglichkeiten für den Entwurf der Klänge. Sie könnten aus einem einzigen Ton bestehen. Um diese Klänge dann voneinander zu unterscheiden, bleiben noch Tonhöhe und Timbre (also das Musikinstrument, das ihn spielt). Allerdings ist die Menge geeigneter Instrumente sehr begrenzt. Zum einen müssen sie von jedem Nutzer korrekt erkannt werden können, d.h. er muss den Klang des Instrumentes kennen, und zum anderem gibt es in vielen Instrumentengruppen starke Ähnlichkeiten zwischen den Instrumenten (z.B. Violine, Viola, Cello, o.ä.).

Die Klänge nur durch die Verwendung verschiedener Tonhöhen zu unterscheiden erscheint ebenfalls problematisch, da man als Nutzer keinen Menschen mit absolutem Gehör voraussetzen kann. Die Unterschiede in den Tonhöhen müssen also groß genug, sein damit sie von jedem Nutzer wahrgenommen werden können, und Klänge nicht verwechselt werden. Hierzu gibt es aber kein genaues Maß, also wären Tests nötig.

Es scheint also sinnvoll, für die Klänge eine kurze Melodie zu nehmen. So hat man nicht nur verschiedene Tonhöhen, sondern auch noch die Elemente Tonart und Rhythmus. Dadurch erhöhen sich die Möglichkeiten zur Erschaffung verschiedener, bzw. leicht unterscheidbarer Klänge immens. Auch kann man sich Melodien sehr gut merken.

Was nun noch zu klären bleibt, ist der genaue Aufbau der Klänge, auch im Hinblick auf Erweiterbarkeit. Will man später noch andere Ereignisse mit Klängen versehen, sollte deren Umsetzung mit dieser vereinbar sein. Ein hierarchischer Ansatz wie bei den *Ear-*

cons oder den Klängen von *CAITLIN* bietet sich auch hier an. Ein solcher Ansatz kann z.B. so aussehen: Aktivitäten bzw. Ereignisse werden durch Klänge dargestellt. Das ist sozusagen die oberste, allgemeine Ebene. Dann bedarf es einer weiteren Ebene, die das Ereignis näher beschreibt, in diesem Fall das Schalten einer Transition. Damit ist aber die Quelle der Aktivität, also um welche Transition es sich handelt, noch nicht beschrieben. Dafür braucht man eine weitere Ebene.

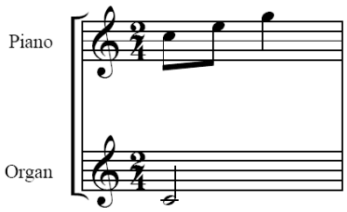

Abbildung 3.1: Beispielklang nach dem entworfenen Schema

Die erste Ebene ergibt sich aus dem Abspielen des Klanges selbst. Auf die dritte Ebene gibt es eine eindeutige Melodie. Es fehlt nur noch ein Element, das die Transitionen als solche identifiziert. Hier bietet sich ein Grundton mit einem bestimmten Timbre an, der zusammen mit der Tonfolge erklingt und über ihre Dauer ausgehalten wird. In Abbildung 3.1 ist der Aufbau eines Klanges in Notenschrift dargestellt. Ein Orgelton der Höhe c‘ identifiziert den Typ Transition und eine kurze Piano-Melodie legt fest, um welche Transition es sich handelt.

3.3 Speicherung des Tonmaterials

Als nächstes muss geklärt werden, auf welche Art die Klänge mit den Transitionen assoziiert werden. Eine einfache Möglichkeit ist, die Klänge als Audiodateien zu speichern und bei Bedarf zu laden und abzuspielen. Dies bietet auch den Vorteil, dass der Nutzer einfachen Zugriff auf die Dateien hat und z.B. für bestimmte Anwendungen eigene Dateien hinzufügen kann.

Denkbar wäre auch die Generierung von Klängen direkt in der Software. Dies brächte allerdings einen erheblich größeren Implementierungsaufwand mit sich, besonders wenn man dem Nutzer auch einige Konfigurationsmöglichkeiten bieten möchte, sodass der Ansatz – auch aus technischen Gründen (siehe Kapitel 4.2.1) – vernachlässigt wurde. Insofern scheint die Verwendung von Audiodateien hier geeignet.

Ein wichtiges Ziel ist, dass die Software zu jeder Zeit nur soviel Speicher nutzt wie nötig. Deshalb wurde davon abgesehen, die Transitionen direkt um ein Audiodatei-Attribut zu versehen. Vielmehr sollten die Transitionen einen Verweis auf den Pfad zur jeweiligen Datei beinhalten, sodass sie erst bei Bedarf, d.h. zur Animation geladen wird, und der dafür benötigte Speicher danach wieder freigegeben werden kann.

Weiterhin ist wünschenswert, dass gespeicherte Netze auch auf anderen Computern mit der Software ausgeführt werden können. Hieraus ergibt sich ein Problem, das aus der Verwendung von Dateien resultiert: Speichert man in den Transitionen den Pfad zu den Dateien, müssen diese sich auf jedem System am gleichen Ort befinden. Daher ist es sehr problematisch die kompletten Pfade zu speichern. Aus diesem Problem ergab sich die Idee einer Audio-Bibliothek, eines Verzeichnisses in dem sich alle Dateien befinden. Dies muss dann lediglich einmal als globale Softwareeinstellung gesetzt werden. Als Konsequenz benötigt man dann in den Transitionen nur noch den Dateinamen (eventuell mit Unterverzeichnis).

Dem gegenüber stand die Idee, die Dateien im gleichen Verzeichnis wie das Netz selbst zu speichern, was aber neue Probleme schafft. Zum einen befänden sich die Dateien dann eventuell auf dem ganzen System verteilt und zum anderen muss man die Dateien, die zur Verwendung zur Verfügung stehen (also noch nicht benutzt werden) trotzdem irgendwo aufbewahren. Sie müssen also dem Programm oder dem Nutzer bekannt sein. Mit der Audio-Bibliothek hat man dagegen einen zentralen Ort, an dem sich die Dateien befinden, was auch der Übersicht über die verfügbaren Klänge zuträglich ist.

4 Umsetzung

Der entworfene Petri-Netz-Typ soll nun in einer bestehenden Softwareumgebung implementiert werden. Dazu werden im Folgenden die Software vorgestellt und die nötigen Anpassungen diskutiert. Ein weiterer Teil der Umsetzung ist die Erstellung entsprechenden Tonmaterials, welches von der Software verwendet werden soll.

4.1 Snoopy Editor

Snoopy ist eine Software zum Modellieren von Graphen, insbesondere von Petri-Netzen. Sie wurde am Lehrstuhl *Datenstrukturen und Softwarezuverlässigkeit* der BTU entwickelt. *Snoopy* unterstützt hierarchische Netze und bietet automatisiertes Layout, also automatisches Anordnen der Netzelemente nach bestimmten Kriterien, für eine bessere Übersicht. Weiterhin können Petri-Netze animiert werden und es bietet die Möglichkeit zum In- und Export von Netzen von bzw. zu einer Reihe anderer Analysewerkzeuge. Für weiterführende Informationen zu Funktionen sei auf [Hei08] verwiesen.

Snoopy ist in der Programmiersprache *C++* implementiert und wird für Windows-, Linux- und Mac-Plattformen unterstützt. Es baut auf dem *wxWidgets*-Framework auf. Sein generisches Design ermöglicht eine einfache Erweiterung um neue Netzklassen und andere Funktionen. Weitere Informationen zur Implementierung finden sich in [Fie04].

4.2 Implementierung

Die nötigen Erweiterungen an der Software lassen sich grob in zwei Bereiche gliedern. Dies sind einerseits die grundlegenden Funktionalitäten, wie die Umsetzung des Musik-Netz-Typs und dessen Animation (in der das Abspielen des Audio zum Tragen kommt) und andererseits Anpassungen und Erweiterungen der Benutzerschnittstelle zur Anwendung der Funktionen. Diese umfassen die Menüs und Kontrollelemente zum Verwalten und Zuweisen von Klängen und zur Animation des Netzes. Ein weiterer wichtiger Teil der Umsetzung ist allerdings die Wahl eines geeigneten Audiodatei-Formats und einer plattformübergreifenden Klassenbibliothek, die das Abspielen von Audio ermöglicht.

4.2.1 Audiodatei-Format und Klassenbibliothek

Da die Dateien von der Software nur abgespielt und nicht manipuliert werden müssen, sind die Anforderungen an Dateiformat und Klassenbibliothek relativ gering. Von der Klassenbibliothek wird verlangt, dass sie plattformübergreifend ist und dass man mehrere Dateien gleichzeitig abspielen kann, falls mehrere Transition zeitgleich, bzw. nahezu zeitgleich schalten, sodass sich sie Klänge bei der Wiedergabe überlappen. Die Klänge sind zwar sehr kurz, aber dennoch ist es von Vorteil, ein Dateiformat zu wählen, bei dem die Dateien so wenig Speicher wie nötig beanspruchen. Aus diesem Grund bietet sich das MIDI-Format an. MIDI (Musical Instrument Digital Interface) ist ein Protokoll zur Übertragung von Musik- und Steuerinformationen zwischen Musikinstrumenten, z.B. die Dauer, die Tonhöhe und das Timbre eines Tones. Empfängt ein MIDI-Gerät eine solche Information setzt es diese als gewünschten Ton um. MIDI-Signale enthalten also zunächst keine Informationen darüber wann sie abzuspielen sind, sondern werden vom Empfänger umgesetzt wenn ihn das Signal erreicht und er bereit ist. Um MIDI-Informationen in Dateien speichern zu können, werden sie um eine Zeitkomponente erweitert, die angibt, wann etwas abgespielt werden soll. MIDI-Dateien sind also sehr klein, da sie nicht die Musik im eigentlichen Sinne enthalten, sondern nur Steuerinformationen. Das führt allerdings auch dazu, dass der Klang von MIDI-Musik von den MIDI-Geräten abhängt, mit denen sie abgespielt wird.

Der entscheidende Grund, der dazu geführt hat, in dieser Arbeit kein MIDI zu verwenden, waren jedoch Probleme eine geeignete Bibliothek zu finden. Es wurde eine Reihe von Bibliotheken betrachtet. Die *Portmidi*- [Por08] und *wxMidi*-Bibliotheken [Sal08] unterstützen nur Echtzeit-MIDI, also MIDI-Signale ohne Zeitinformationen und eignen sich daher nicht, um MIDI-Dateien abzuspielen. Die *Allegro*-Bibliothek [All08] unterstützt zwar MIDI-Dateien, erlaubt aber nur jeweils eine Datei auf einmal abzuspielen. Da keine geeignete MIDI-Bibliothek gefunden werde konnte, blieb nur der Ausweg zu einem anderen Dateiformat.

Hier fiel die Wahl auf die *irrKlang*-Bibliothek [Amb08] der österreichischen Firma *Ambiera.* Sie unterstützt die Dateiformate *RIFF WAVE* (.wav), *Ogg Vorbis* (.ogg), *MPEG-1 Audio Layer 3* (.mp3), *Amiga Modules* (.mod), *Fast Tracker 2* (.xm), *Scream Tracker 3* (.s3d) und *Impuls Tracker* (.it) und bietet außerdem ein sehr einfaches aber umfangreiches API. Die von der Bibliothek unterstützten Tracker-Formate haben gewisse Ähnlichkeiten zu MIDI-Dateien und damit auch einige ihrer Vorteile: Sie sind klein und einfach zu erstellen, werden aber heute kaum noch unterstützt, weshalb die Wahl dann auf das weit verbreitete WAVE-Format fiel.

Diese Entscheidung bringt allerdings zwei Nachteile mit sich. Einerseits werden die Dateien wesentlich größer als MIDI-Dateien. Wichtiger ist jedoch, dass man sich als Nutzer der Software relativ leicht selbst Klänge im MIDI-Format erstellen kann, was bei dem WAVE-Audioformat nicht so einfach geht. So gibt es eine Reihe von MIDI-Editoren (z.B.

Anvil Studio [Anv08]) mit denen man die MIDI-Musik über ein Notensystem oder eine Klaviatur einfach am Computer eingeben kann, während man sonst die Musik mit richtigen Instrumenten aufnehmen müsste. Im Hinblick auf mögliche zukünftige Weiterentwicklungen dieser Arbeit ist folgender Kompromiss entstanden: Die Umsetzung der Klänge erfolgt wie ursprünglich geplant als MIDI-Dateien. Damit diese trotzdem genutzt werden können, werden sie in das unterstützte WAVE-Format konvertiert. So liegen falls nötig die Klänge auch im MIDI-Format vor.

4.2.2 Netzklasse und Animation

Bei dem entworfenen Netz-Typ handelt es sich um ein um spezifische Eigenschaften erweitertes Petri-Netz. Die Architektur von Snoopy erlaubt, diesen Umstand direkt umzusetzen, indem man die hier benötigte Netzklasse von der normalen Petri-Netzklasse ableitet und den Transitionen eine weitere Eigenschaft gibt. Wie bereits erwähnt, handelt es sich dabei um ein Textattribut, das den Pfad zu der entsprechenden Audiodatei beinhaltet. Umgesetzt wurde dies letztlich als Listenattribut, d.h. es können einer Transition mehrere Klänge zugeordnet werden. Da jede Transition über eine solche Liste (gleicher Länge) verfügt, erhält man die Möglichkeit, zur Animation einen Listenindex auszuwählen und so die Audioanimation auf eine bestimmte Zielstellung hin anzupassen. Man kann also zur Animation festlegen, welche der verfügbaren Klänge verwendet werden sollen.

Die Animation wird in einer eigenen Klasse organisiert, die die Animation eines typischen Petri-Netzes um die notwendigen Eigenschaften zum Abspielen, Laden und Löschen von Audio erweitert. Snoopy bietet zwei Modi bei der Arbeit mit Petri-Netzen: Den Editiermodus zum Erstellen und Bearbeiten und den Animationsmodus für die Animation mit den Marken, in dem das Netz nicht editiert werden kann. Sobald man den Animationsmodus startet, werden die Sounds geladen und beim Beenden des Modus wieder gelöscht, um Geschwindigkeit und Speicherbedarf zu optimieren.

4.2.3 Benutzerschnittstelle

Um die Änderungen nutzbar zu machen, mussten eine Reihe von Fenstern und Menüs angepasst und hinzugefügt werden. Diese umfassen zum Beispiel die Kontrolle über die Animation oder die Zuweisung von Klängen zu Transitionen. Einige Abbildungen zur Umsetzung befinden sich in Anhang A.

4.2.4 Automatisches Zuordnen von Klängen

Obwohl die Möglichkeit, Klänge jeder Transition manuell zuzuordnen, notwendig und sinnvoll ist, kann dies abhängig von der Größe des Netzes schnell mühsam werden. Es ist also eine Funktion gefragt, die dem Nutzer diese Arbeit abnimmt. Da aber die zufällige Vergabe von Klängen wenig zweckmäßig ist, ergab sich die Idee, Klänge nach bestimmten Kriterien oder Algorithmen zu vergeben. Je nach Anwendungsfall können diese sehr verschieden sein. Darum wurde in dieser Arbeit ein Kriterium beispielhaft implementiert: Die Zuordnung von Klängen zu Transitionen anhand der Netzstruktur.

Transitionen lassen sich anhand der Anzahl ihrer ein- und ausgehenden Kanten und ihrer Gewichte in markenerhaltende, -konsumierende und -produzierende Transitionen einteilen. Um diese Eigenschaften festzustellen, wurde ein Algorithmus implementiert, der den Graph durchmustert und die Kanten und deren Gewichte an den Transitionen zählt. Nachdem die Transitionen in diese drei disjunkten Mengen aufgeteilt wurden, bekommen diese jeweils eine Art von Klängen zugewiesen. Dazu wurden Klänge erstellt, deren Melodien sich im Timbre unterscheiden. Diese sind im Einzelnen Gitarre für markenerhaltende, Trompete für markenkonsumierende und Flöte für markenproduzierende Transitionen.

Die Funktion dient in dieser ersten Implementierung nur als Beispiel und dazu, die Rahmenstruktur (z.B. die Menüs) für weitere Algorithmen zu erstellen. Allerdings ist sie schon ein gutes Beispiel für den Nutzen von Audio, denn hier hat man die Möglichkeit, durch die Animation mit Audio auf simple Art Informationen über die Netzstruktur zu bekommen. Besonders im Hinblick auf Softwaretests sind hier noch weitere Verfahren denkbar. Diese Funktion zum automatischen Zuordnen von Klängen ist potentiell die wichtigste bei der Erweiterung der Petri-Netze um akustische Eigenschaften, denn sie ermöglicht es Eigenschaften des Netzes während der Animation zu verdeutlichen, die sonst zu diesem Zeitpunkt nicht so offensichtlich wären.

5 Auswertung

Im Rahmen dieser Arbeit konnte das Thema noch nicht erschöpfend bearbeitet werden. Somit ist eine abschließende Beurteilung zu diesem Zeitpunkt noch nicht möglich. Neben den bisher erfolgten Funktionstests sind noch weitere Tests nötig. Es gilt zu ermitteln, ob und inwiefern die akustische Erweiterung von Snoopy für den Anwender tatsächlich einen Mehrwert darstellt und ihn bei der Arbeit unterstützt. Ausführliche Anwendungstests sollten durchgeführt werden, um dies beurteilen zu können.

Dies trifft in gewissem Maße auch auf die Benutzeroberfläche in Bezug auf Usability zu. Zwar wurde versucht, die neuen Elemente so umzusetzen, dass die Benutzung intuitiv und vor allem konsistent zur bisherigen Umsetzung ist, aber sie sind noch nicht hinreichend getestet.

Insbesondere muss untersucht werden, welche Funktion beim Testen und Validieren von Software noch benötigt werden, denn die jetzige Umsetzung ist noch nicht sehr softwarespezifisch, d.h. das Anwendungsgebiet beschränkt sich nicht nur auf Software. So ist ein Experimentieren in anderen Anwendungsbereichen auch möglich. Der wichtigste Punkt, der noch Aufmerksamkeit bedarf, ist die Funktion zur automatischen Zuordnung von Klängen. Hier muss geprüft werden, welche Algorithmen noch nötig sind bzw. hilfreich sein können.

Außer den Klängen sind noch weitere Funktionen und Verbesserungen wünschenswert. So wäre zum Beispiel eine zukünftige Unterstützung von MIDI-Dateien geeignet, um die Größe der Dateien zu verringern und dem Nutzer das Erstellen eigener Klänge zu vereinfachen. Die jetzige Lösung mit WAVE-Dateien ist in diesem Zusammenhang eher als Kompromiss anzusehen, der gewählt wurde, um durch die Suche nach einer geeigneten MIDI-Bibliothek die Arbeit nicht weiter aufzuhalten.

Es sollten auch noch mehr Klänge erstellt werden, damit man auch größere Graphen entsprechend bedienen kann. Sinnvoll wäre es auch, Netzhierarchien besser zu verdeutlichen. Dafür könnte man den Klang einer schaltenden Transition in Abhängigkeit der Hierarchieebene durch Effekte, wie z.B. Hall verändern bzw. jedem Teilnetz einen eigenen Effekt zuweisen.

Weitere wichtige Funktionen die Snoopy für seine bisherigen Netztypen anbietet, sind Konvertierungsmöglichkeiten zu anderen Netztypen, sowie Im- und Exportfunktionen zu anderen Analyseprogrammen. Diese wurden in der jetzigen Implementierung noch nicht umgesetzt, sollten aber konsequenterweise auch für das Musik-Petri-Netz zur Verfügung stehen.

Sollte sich das Musik-Petri-Netz in der Praxis bewähren, könnte man in Betracht ziehen, die Musik-Eigenschaft auch in erweiterte Petri-Netze einzubauen.

Glossar

Akkord Das gleichzeitige Erklingen mehrerer Töne, die in einem harmonischen Zusammenhang stehen, z.B. in Terzintervallen geschichtet sind.

Atonalität Bezeichnet Musik, die nicht auf einem Zentrum oder Grundton beruht, sodass alle Töne die gleiche Bedeutung haben. *Vgl. Tonalität*

Invertierung Ein Kompositionsmittel, bei dem eine bestehende Melodie inklusive aller Notenlängen rückwärts gespielt oder notiert wird, um eine neue Melodie zu erhalten.

Klang Als Klang wird im Kontext dieser Arbeit ganz allgemein ein akustisches Signal verstanden. Dies kann eine Tonfolge oder eine Melodie sein, muss aber nicht zwangläufig musikalischer Natur sein.

Melodie Eine Tonfolge, deren Töne auch einen Rhythmus haben, die also auch verschiedene Tondauern haben können.

Rotation Ein Kompositionsmittel, bei dem eine bestehende Melodie in mehrere (zeitliche) Einheiten zerlegt wird, (z.B. taktweise), die dann nach dem Rotationsprinzip neu aneinander gereiht werden, z.B. indem der erste Takt nach hinten gesetzt wird. So können in mehreren Schritten neue Melodien erzeugt werden.

Spiegelung Ein Kompositionsmittel, bei dem eine Melodie an einem bestimmten Referenzton in der Tonhöhe gespiegelt wird. Auf diese Weise wird z.B. ein Ton, der eine kleine Terz unter dem Referenzton steht, eine kleine Terz über den Referenzton bewegt.

Timbre Ein gespielter Ton eines Musikinstruments besteht neben dem gespielten Grundton aus vielen Obertönen und Rauschanteilen. Sie bestimmen das Timbre (die Klangfarbe) des Tones und des ganzen Instruments und bewirken, dass man z.B. eine Violine von einer Trompete unterscheiden kann. Vereinfacht gesagt, bestimmt das Timbre das Musikinstrument, weswegen die Begriffe häufig synonym verwendet werden.

Tonalität Bezeichnet Musik, deren Töne im Bezug zu einem Zentrum oder Grundton stehen. Daraus ergeben sich Rangordnungen innerhalb dieses Systems und die Töne und Akkorde, die sich aus ihnen bilden lassen, bekommen eine bestimmte Funktion zugeschrieben.

Tonfolge Eine Tonfolge sind im Kontext dieser Arbeit Töne, die in einem musikalischen Zusammenhang (z.B. einer Tonart) stehen. Alle Töne haben die gleiche Dauer.

Transposition Ein Kompositionsmittel, bei dem ein Ton oder eine Melodie anhand eines festen Intervalls in der Höhe verschoben wird.

Literaturverzeichnis

[All08] Allegro. A game programming library. [Online] [Zitat vom: 18. Mai 2008.] http://www.talula.demon.co.uk/allegro/.

[Alt95] Alty, James L. 1995. Can We Use Music in Computer-Human Communication? *People and Computers X.* s.l. : Cambridge University Press, 1995.

[Amb08] Ambiera. irrKlang. [Online] [Zitat vom: 18. Mai 2008.] http://www.ambiera.com/irrklang/index.html.

[Anv08] Anvil Studio. *MIDI-Editor.* [Online] [Zitat vom: 18. August 2008.] http://www.anvilstudio.com/.

[Bre98] Brewster, Steven A. 1998. *The design of sonically-enhanced widgets. Interacting with Computers.* 1998.

[Edw95] Edwards, Alistair D. N. 1995. The Rise of the Graphical User Interface. [Online] Dezember 1995. [Zitat vom: 14. Mai 2008.] http://people.rit.edu/easi/itd/itdv02n4/article3.htm.

[Fie04] Fieber, Markus. 2004. *Entwurf und Implementierung eines generischen, adaptiven Werkzeugs zur Arbeit mit Graphen. Diplomarbeit.* Cottbus : BTU Cottbus, 2004.

[Hau98] Haunschild, Frank. 1998. *Die neue Harmonielehre.* Brühl : AMA-Verlag, 1998. 3-927190-00-4.

[Hau92] —. 1992. *Die neue Harmonielehre 2.* Brühl : AMA-Verlag, 1992. 3-927190-08-X.

[Hei08] Heiner, Monika, et al. 2008. Snoopy - A Tool to Design and Execute Graph-Based Formalisms. *Petri Net Newsletter.* 2008, 74.

[DSSZ08] Lehrstuhl Datenstrukturen und Softwarezuverlässigkeit, BTU Cottbus. 2008. Snoopy. [Online] August 2008. [Zitat vom: 11. August 2008.] http://www-dssz.informatik.tu-cottbus.de/software/snoopy.html.

[Lev95] Levens, Ursula M. 1995. *Computergestütztes Modellieren von Musikstücken mit Petri-Netzen: Das Mailänder Konzept.* Universität Oldenburg : s.n., 1995.

[Mey08] Meyers, Scott. 2008. *Effektiv C++ programmieren.* s.l. : Addison-Wesley, 2008. 978-3827326904.

[Pet08] Petri, Carl Adam und Reisig, Wolfgang. 2008. Petri net. [Online] 1. Februar 2008. [Zitat vom: 9. August 2008.] http://www.scholarpedia.org/article/Petri_net.

[Por08] Portmedia. *Platform Independent Libraries for Sound and MIDI.* [Online] [Zitat vom: 22. Mai 2008.] http://portmedia.sourceforge.net/.

[Rei86] Reisig, Wolfgang. 1986. *Petrinetze.* Berlin : Springer-Verlag, 1986. 3-540-16622-X.

[Sac07] Sacks, Oliver. 2007. Oliver Sacks on Earworms, Stevie Wonder and the View From Mescaline Mountain. *Wired Magazine.* 2007, 15.10.

[Sal08] Salmeron, Cecilio. wxMidi. [Online] [Zitat vom: 22. Mai 2008.] http://wxcode.sourceforge.net/components/wxmidi/.

[Sta90] Starke, Peter. 1990. *Analyse von Petri-Netz-Modellen.* Stuttgart : Teubner, 1990. 3-519-02244-3.

[Str00] Stroustrup, Bjarne. 2000. *Die C++ Programmiersprache.* s.l. : Addison-Wesley, 2000. 978-3827316608.

[Vic08] Vickers, Paul. Auditory External Representations of Programs. [Online] [Zitat vom: 10. Juni 2008.] www.auralisation.org.

[Vic99] —. 1999. *CAITLIN: Implementation of a Musical Program Auralisation System to Study the Effects on Debugging Tasks as performed by novice Pascal programmers.* Loughborough University, Loughborough UK : s.n., 1999.

[Vic03] Vickers, Paul und Alty, James L. 2003. Siren Songs and Swan Songs. *Communications of the ACM.* 2003, 46/July 2003.

[Wil01] Willms, André. 2001. *C++ Programmierung.* s.l. : Addison-Wesley, 2001. 978-3827322975.

A Benutzeroberfläche

Nachfolgend sind einige Dialoge dargestellt, die angepasst wurden.

Abbildung A.1: Animationskontrollfenster bei normalen Petri-Netzen

Abbildung A.2: Animationskontrollfenster bei Musik-Petri-Netzen: Hier kann man zusätzlich zu A.1 das zu verwendende Tonmaterial auswählen und bei Bedarf die Audiowiedergabe deaktivieren

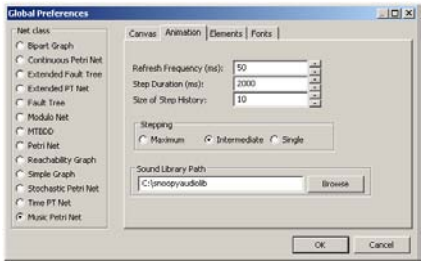

Abbildung A.3: Globale Programmeinstellungen

In den globalen Programmeinstellungen (Abbildung A.3) kann man den Speicherort der Audio-Bibliothek angeben.

Abbildung A.4: Eigenschaften-Menü einer Transition

Abbildung A.4 zeigt das Eigenschaften-Menü einer Transition eines Musik-Netzes.
Unten werden die zur Verfügung stehenden Standardklänge aufgelistet. Um der Transition einen Klang zuzuweisen, wählt man den Klang aus der Liste aus, markiert die entsprechende Zelle in der Tabelle und klickt den *Select*-Button.

Alternativ kann man mit *Apply Sound by Algorithm* Klänge automatisch zuweisen.

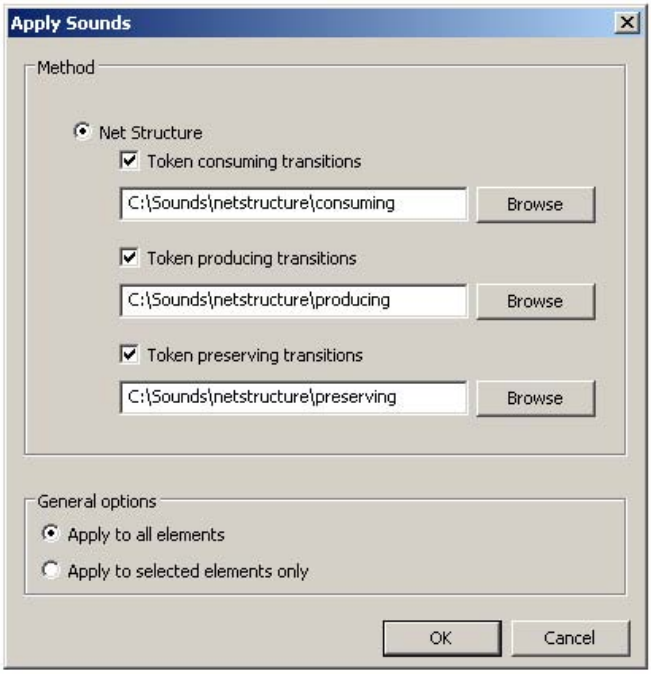

Abbildung A.5: Automatisches Zuweisen

Abbildung A.5 zeigt den Dialog zum automatischen Zuweisen von Klängen.
Derzeit wird nur der Algorithmus zum Zuweisen anhand der Netzstruktur angeboten. Für jede Transitionsart kann ein eigenes Verzeichnis mit Klängen gewählt werden.

B Aufbau der Audio-Bibliothek

Abbildung B.1 zeigt den Aufbau der Audio-Bibliothek. Im Hauptverzeichnis *snoopyaudiolib* befinden sich die Standardklänge, die manuell mit Transitionen assoziiert werden können. Im Verzeichnis *netstructure* bzw. dessen Unterverzeichnisse sind die Dateien, die für den implementierten Algorithmus zum automatischen Zuordnen anhand der Netzstruktur entworfen wurden.

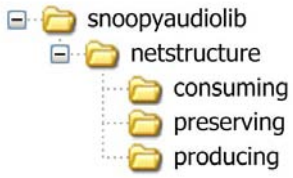

Abbildung B.1: Verzeichnisstruktur der Audio-Bibliothek

Die Standardklänge bestehen aus einer kurzen Melodie mit Piano-Timbre über einen Orgelgrundton. Die Klänge für die Netzstruktur unterscheiden sich von ihnen im Timbre der Melodie. Dies sind für die Kategorien markenerhaltend, markenkonsumierend und markenproduzierend jeweils Gitarre, Trompete und Flöte.