UNIVERSITY OF
Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

UNIVERSITY OF SOUTHAMPTON

Motivated Music: Automatic Soundtrack Generation for Film

by

Michael O. Jewell

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

February 2007

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

MOTIVATED MUSIC: AUTOMATIC SOUNDTRACK GENERATION FOR FILM

by Michael O. Jewell

Automatic music composition is a fast-moving field which, from roots in serialism, has developed techniques spanning subjects as diverse as biology, chaos theory and linguistic grammars. These algorithms have been applied to specific aspects of music creation, as well as live performances. However, these traditional approaches to generation are dedicated to the creation of music which is independent from any other driving medium, whereas human-composed music is most often written with a purpose or situation in mind. Furthermore, the process of composition is naturally hierarchical, whereas the use of a single algorithm renders it a monolithic task.

In order to address these issues, a model should be able to encapsulate a sense of composer motivation whilst not relying on a single algorithm for the composition process. As such, this work describes a new framework with the ability to provide a means to generate music from film in a media-driven, distributed, manner. This includes the initial annotation of the media using our new OntoMedia ontology; the mapping of annotated information into parameters suitable for compositional algorithms; the design and implementation of an agent framework suitable for the distribution of multiple composing algorithms; and finally the creation of agents capable of handling the composition of musical elements such as rhythm and melody. In addition, a case study is included which demonstrates the stages of the composition process from media annotation to automatic music generation.

Contents

Acknowledgements x

1 Introduction 1
- 1.1 The State-Based Sequencer . . . 3
- 1.2 Contributions . . . 5

2 The Traditional Composition Process 7
- 2.1 Filming . . . 7
- 2.2 Screening . . . 8
- 2.3 Spotting . . . 8
- 2.4 Laying Out . . . 9
- 2.5 Motif Development . . . 9
- 2.6 Final Composition . . . 10
- 2.7 Summary . . . 10

3 Automating Composition 11
- 3.1 Stochastic . . . 12
- 3.2 Cellular Automata . . . 13
  - 3.2.1 Demon Cyclic Space . . . 14
  - 3.2.2 Composing with Cellular Automata . . . 14
    - 3.2.2.1 CAMUS . . . 15
- 3.3 Grammatical Production . . . 16
  - 3.3.1 Composing with Grammatical Productions . . . 16
- 3.4 Fractal . . . 17
  - 3.4.1 Composing with Fractals . . . 17
- 3.5 Genetic Algorithms . . . 18
  - 3.5.1 Composing with Genetic Algorithms . . . 18
- 3.6 Summary . . . 19

4 Media Annotation 20
- 4.1 Describing Media . . . 20
- 4.2 Video Analysis . . . 21
  - 4.2.1 Colour Detection . . . 21
  - 4.2.2 Motion Detection . . . 22
  - 4.2.3 Transition Detection . . . 23
  - 4.2.4 Cast Member Identification . . . 24
- 4.3 Audio Analysis . . . 24
  - 4.3.1 Vocal Detection . . . 24

4.3.2 Music Detection . . . 24
4.3.3 Foley Detection . . . 24
4.4 Script Annotation . . . 25
4.5 Summary . . . 26

5 The OntoMedia Ontology 27
5.1 A Brief Introduction to Ontologies . . . 27
5.2 The Structure of OntoMedia . . . 28
5.3 The Ontology . . . 29
5.3.1 Entity Modelling . . . 31
5.3.2 Event Modelling . . . 32
5.4 Extensibility . . . 33
5.4.1 Names . . . 33
5.4.2 Geometry . . . 34
5.5 Case Studies . . . 34
5.5.1 Applying to Fiction . . . 34
5.5.2 Applying to Film . . . 34
5.6 Summary . . . 35

6 Composer Representation 37
6.1 Musical Modifiers . . . 37
6.1.1 Tempo . . . 38
6.1.2 Pulse . . . 39
6.1.3 Rhythm . . . 39
6.1.4 Scale and Key . . . 40
6.1.5 Chord . . . 41
6.1.6 Instrumentation . . . 42
6.1.7 Melody . . . 43
6.2 Binding to a Semantic Annotation . . . 44
6.3 Summary . . . 45

7 The Agent Framework 46
7.1 Introduction . . . 46
7.2 Agent Design . . . 46
7.2.1 Identification . . . 48
7.2.2 Ports . . . 48
7.2.3 Monitors . . . 49
7.3 Router Design . . . 49
7.4 Agent Graphs . . . 50
7.5 Agent Launcher . . . 51
7.6 Summary . . . 51

8 Agent Designs 52
8.1 Overview . . . 52
8.1.1 From Algorithm to Agents . . . 52
8.1.2 Designing Agents for Musical Composition . . . 53
8.1.2.1 Specialising for Genetic Algorithms . . . 54
8.1.2.2 Philosophy . . . 55

8.2 Tempo Agent . . . 56
8.3 Pulse Agent . . . 57
8.4 Key Agent . . . 58
8.5 Chord Agent . . . 60
8.6 Instrumentation Agent . . . 60
8.7 Rhythm Agent . . . 61
8.8 Melody Agent . . . 62
8.9 Summary . . . 64

9 A Case Study 65
9.1 Screenplay Annotation . . . 65
9.2 Location Annotation . . . 67
9.3 Character Annotation . . . 67
9.4 Event Annotation . . . 68
9.5 Querying the OntoMedia Representation . . . 70
9.5.1 All People . . . 71
9.5.2 Linking Actors to Characters . . . 71
9.5.3 Locating Specific Events . . . 72
9.6 Creating the Landmark Representation . . . 72
9.6.1 Location: Mars . . . 72
9.6.2 Characters: Quaid and Melina . . . 73
9.6.3 Character: Lori . . . 74
9.6.4 Interactions . . . 75
9.6.5 Generating the Landmark File . . . 76
9.7 Composing The Music . . . 76
9.7.1 Tempo Agent . . . 77
9.7.2 Pulse Agent . . . 77
9.7.3 Key Agent . . . 77
9.7.4 Chord Agent . . . 78
9.7.5 Instrumentation Agent . . . 81
9.7.6 Rhythm Agent . . . 81
9.7.7 Melody Agent . . . 81
9.8 Evaluation . . . 82
9.9 Summary . . . 83

10 Conclusions 87
10.1 Overall Conclusions . . . 87
10.2 Future Work . . . 88
10.3 Top-Down Composition . . . 89

References 90

A The OntoMedia Core Ontology 94

List of Figures

1.1 The Structure of the State-Based Sequencer 4

3.1 Three stages of a Game of Life cellular automata. This configuration is known as a 'lightweight spaceship' (LWSS) which progresses along the lattice over subsequent iterations. 14
3.2 Two figures showing the development of a Demon Cyclic Space. Beginning with a random configuration, (a) shows the initial signs of domination, while (b) shows the development of stable spiral patterns. 15

4.1 The use of the colour red to represent a dream in David Lynch's *Twin Peaks*. 21
4.2 Two characters falling onto a building in *The Matrix*. 22
4.3 The edge change information computed for a segment from *The Matrix*. The peaks represent shot transitions present within the sequence. 23

5.1 The class structure of the OntoMedia ontology 29
5.2 The core and extension modules within OntoMedia, with the classes of the core modules listed. 30
5.3 The OntoMedia miscellaneous modules. These do not rely on the core or extension classes. 30
5.4 The OntoMedia entity model 31
5.5 The OntoMedia Timeline structure 32
5.6 The OntoMedia event model 33
5.7 Binding OntoMedia objects to other media 35

6.1 C major scale shown using notes of a chromatic scale. 40
6.2 Mapping from a composer representation to a landmark file 45

7.1 An example agent graph, with numbers indicating the order of execution given identical agent execution times. 50

8.1 The standard agent template employed by SBS 53
8.2 The SBS Agent Arrangement 54
8.3 Output from the Pulse Agent 58
8.4 An example key graph. Note the probabilities of transitioning from one key to another. 59
8.5 Output from the Rhythm Agent 62
8.6 Output from the Melody Agent 63

9.1 Meditate: Entry for Character "Douglas Quaid" 68

9.2 A portion of the occurrences in the Total Recall timeline. 69
9.3 The melody generated for the first segment. 85

List of Tables

6.1 Approximate BPM ranges for standard tempos. 38

7.1 The message types available to the Light Agent Framework. All return OK or NOK on success or failure. 47

8.1 A simple instrument/weighting mapping for a segment. 61

Listings

4.1 An annotated excerpt from Apocalypse Now 26
5.1 Example queries in SeRQL, SPARQL, and RDQL respectively. 28
6.1 An example tempo modifier 39
6.2 An example tempo modifier using a beat placement 39
6.3 An example pulse modifier 39
6.4 An example rhythm modifier 40
6.5 A major scale definition 40
6.6 A definition of C Major 41
6.7 Defining C Major's relative minor 41
6.8 An example chord modifier 42
6.9 The instrument definition for a bowed violin 43
6.10 An example melody modifier 44
6.11 Promoting the use of strings when an event takes place in a corridor. . . . 44
6.12 A portion of a generated landmark file. 45
7.1 The external stub file for a string concatenation agent 48
9.1 A section from *Total Recall* in SiX format 66
9.2 Binding an OntoMedia representation to a SiX screenplay resource 66
9.3 Defining the location of a scene 67
9.4 Meditate RDF for Douglas Quaid 69
9.5 Defining the events within a scene 70
9.6 Defining the occurrences of events 70
9.7 The composer representation for Mars. 73
9.8 The composer representation for Quaid and Melina. 74
9.9 The composer representation for Lori. 75
9.10 A composer representation capable of altering key when guns are lost. . . 76
9.11 The landmark generated for the first shot of the scene. 77
9.12 The tempo landmarks generated for the first shot of the scene. 78
9.13 The MusicXML tempo generated for the first shot of the scene. 78
9.14 The pulse landmarks generated for the first shot of the scene. 79
9.15 The MusicXML time signature and measures generated for the first shot of the scene. 79
9.16 The key landmark generated for the first shot of the scene. 80
9.17 The MusicXML key signature generated for the first shot of the scene. . . 80
9.18 The chord landmarks placed on the strong beats of the segment. 80
9.19 The instrument landmarks placed at the start of the segment. 81
9.20 The MusicXML with additional instrument parts. 82
9.21 The instrument landmarks placed at the start of the segment. 83

9.22 The MusicXML with note durations. 84
9.23 The first set of notes generated by the melody agent. 85
9.24 The initial notes for the string part in MusicXML format. 86

Glossary

aleatoric: Music in which some element of the composition is left to chance or in which an element of the performance is determined by the performer, 12

diegetic: Sound that other characters in a film or play would be able to hear, such as a song on a radio which is present in the scene, 24

homophonic: A piece of music where all voices proceed rhythmically at the same rate, 40

non-diegetic music: Sound whose source is neither visible on the screen nor has been implied to be present in the action, 2

polyphonic: A piece of music where each voice of a composition proceeds independently of the others, 40

Acknowledgements

I am very grateful to my supervisors, Mark Nixon and Adam Prügel-Bennett, for their guidance throughout my research and thesis writing, particularly their resilience to such a broad project and their ability to keep me away from too many tangents. Many thanks are also owed to my collaborators: lee 'lowercase' middleton for his work on the Light Agent Framework; and Faith Lawrence and Mischa Tuffield for their contributions to the OntoMedia ontologies.

Finally, I would like to thank my family and friends for their support and encouragement, and the good people of *#notts* for providing endless alternatives to thesis writing.

Chapter 1

Introduction

Algorithmic composition, or the generation of music using programmatic approaches, has progressed rapidly throughout the last century. From the early serialism experiments, to fractal and cellular techniques, and to the present day with methods analogous to Darwinian evolution. However, in the transition from composer to computer these algorithms lose one of the fundamental aspects of the composition process: a sense of motivation. While early music is now typically performed in concert halls, it is often forgotten that many pieces were written for specific occasions or purposes. A key example of this is music intended for dancing, such as waltzes and burlesques, which were often written to be performed at state gatherings. This has continued to the modern day, although it is now classified as 'dance music'. The rhythm and pulse are essential to this form; an irregular number of beats would confuse dancers, and a tempo which is too fast or slow for the atmosphere would be unsuitable. Dance music composers take this into consideration while writing the pieces: the music is motivated by the composer's intention for the music.

At a more abstract level, the romantic movement (which coincided with the impressionist era of painters) produced some extremely directed music. This was furthered with 'program music', which explicitly endeavoured to represent and accompany extra-musical themes. Pictures at an Exhibition, composed by Modest Mussorgsky in 1874, documented the pictures within a friend's gallery, with each movement separated by a 'promenade' movement to indicate walking through the exhibition. To portray these situations in music, the composer translated elements from the visual domain into the aural. For example, a bulky oxcart was represented by a solo euphonium, a troubador with an alto saxophone, and a man by a piercing, troubled-sounding melody - to suggest his thin, poor, character[1]. In this sense, programme music is a precursor to the film soundtrack - namely a piece which can instill a sense of the surroundings and events taking place, albeit with only music as an indicator.

[1] Russ [1992]

The opera, which originated in 1597 with Jacopo Peri's *Dafne*, can be seen as a more tightly bound approach to film. Rather than the accompanying music acting as a separate entity to the dialogue and action, opera adhered the dialogue directly to the music. In effect this takes the approach of programme music and incorporates the singers as additional instruments. Naturally this is harder to compose, as the vocal ranges and techniques of the performers must be taken into consideration, and the music must be present while not distracting from the words of the piece. It may be argued that opera is in fact closer to film than programme music due to the visual aspect, yet the dialogue in movies is divorced from the accompanying soundtrack and the music is relegated into a secondary position, so film music is more akin to a passive form of programme music than opera.

This suggestion is further justified by examining the origination of film: the silent movie. In movie theatres, an organist would accompany silent films - primarily to disguise the noise of the machinery, but also to provide something for the viewers to listen to while viewing. This was in fact the first evidence of a soundtrack, with the improvisational organ playing providing fast-paced music during chase scenes and sinister undertones when the 'bad guy' made an appearance. In this sense, the organist provided motivation to the music - a motivation guided by the content of the film. When the first talkies with accompanying musical scores (Lights of New York in 1928, and The Squall in 1929) were revealed, the soundtrack was very different to what is expected today. The music was stylized and unrealistic, and as it was recorded live with the actors there was often a battle between the dialogue and the accompaniment. It was due to the criticism that this wrought that Hollywood abolished non-diegetic music in film[2], only allowing 'mood' music from sources within the scene such as radios. Thus, the first 'traditional' soundtrack was in 1932, with the horror film 'Most Dangerous Game, Trouble in Paradise'.

Present-day human-composed music is not so different from that which is deemed 'classical', it is merely the motivations which have varied. As recorded music became preferable to the average listener, the public instead purchased music where the creator's composition technique 'gelled' with theirs. Opera, while still composed, is now available in the more publicly-accessible musicals, while 'dance music' has spread into a spectra of varieties ranging from slow-paced 'lounge' to frenetic rave and garage music. Instrumental music has also transformed, with the African origins of jazz and blues in the years following 1910 leading to significant alterations to traditional scales and harmonies. This is not to say that the waltzes and marches have been superceded, instead the proliferation of new instruments and performing locations has created a branch of the original style. The 'motivation' for music writing is still significant (no matter how much money is involved) and even as circumstances change, the need for direction in the composition of musical works is paramount.

[2] Everson [1978]

Thus, it is evident that besides the technicalities of music writing, the composer also provides a direction to the music they create. Whether the aim is to support text, to put a scene to music, to symbolise the sense of anger of a performer on a stage, or to provoke a reaction from dancers on a dance floor, this intention is missing from existing algorithmic composition techniques: these processes only embody the creation of an independent piece of music.

SBS, or the State-Based Sequencer, therefore comprises an approach to create music which is influenced through the provision of an extra piece of media. Algorithmic principles are still utilised, but the presence of a supporting material allows for the generation of a result which has a sense of direction and form. For this purpose, we focused on film - primarily due to the abundance of information which may be acquired from the media in its many representations (e.g. video, script, and sound effects). SBS is further novel through its use of a 'decomposed' composing strategy. This divides the process into logical components, each handled by individual composing agents.

1.1 The State-Based Sequencer

The SBS strucure is tripartite, with each component focusing on a different perspective of soundtrack composition. The annotation stage could be seen as representing a director's intention in a film, the composer stage representing a composer's perception of themes, and the final stage representing the composer's composing technique when applied to the film. Each of these three areas is described in this thesis, as well as how they have been designed to be analogous to the typical composition process.

The first stage of the SBS process involves the labeling of the film media with elements which are likely to alter the style of the soundtrack. Certain characters may warrant the inclusion of specific instruments or locations may be suggested with the use of particular rhythms or keys. At present, SBS uses a selection of high-level image processing algorithms to initially process the film, splitting the media into segments, determining the median colour of a segment, and calculating the amount of motion within a segment. Further annotation is then carried out using manual techniques, although this process may be alleviated by automatic systems in future. The final annotated version conforms to the OntoMedia Ontology, which is described in Chapter 5.

Following the annotation of the film, a translation from the ontological form to a musical form is required. This is provided in the form of a composer representation, which queries the ontology for certain elements, placing musical modifiers into a collection of parameters. Chapter 6 describes the form of this definition, as well as the full set of parameters which may be specified using the representation.

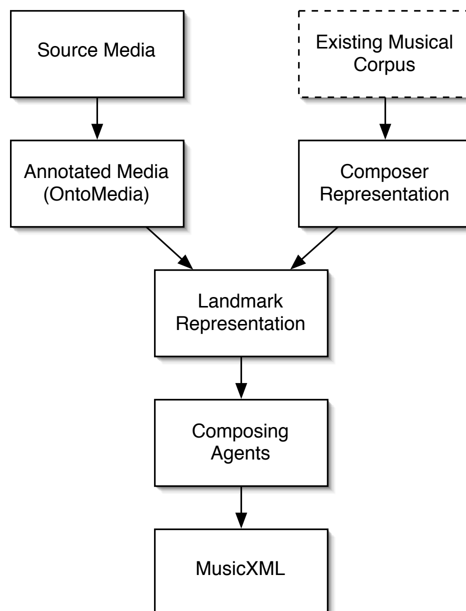


FIGURE 1.1: The Structure of the State-Based Sequencer

The culmination of the SBS system is the composition of the film soundtrack. This relies heavily on the Light Agent Framework, a system developed for SBS and discussed in Chapter 7, to distribute composing agents in a connected graph. This isolates the agents, allowing for individual testing, as well as emulating more accurately the traditional composing cycle. The 'state' element of the State Based Sequencer is key here, as a parameter file containing 'landmark' points is used to indicate the musical state at any one time to the agents. Each of these agents may contain algorithms specific to its requirements, and the algorithms chosen for SBS are described in Chapter 8, along with results from individual tests.

This thesis describes the process of creating music using the SBS system. First, the composition process for film is examined to demonstrate the parallels between our method and traditional approaches. Next, we consider the existing techniques for algorithmic composition, with a focus on how these may be used to good effect in a film composition system. An outline of SBS as a whole is then given and the subsequent chapters examine the sections of SBS from the annotation of the supporting media using a custom-written ontology, to the representation of a composer which may translate from this media to musical modifiers, and finally to the composition of musical elements using a distributed set of composing agents (again implemented in a custom-built framework). To finish, a case study of film music composition is followed through to illustrate how SBS may be applied.

1.2 Contributions

The annotation stage of SBS incorporates OntoMedia. This was designed and implemented by a group of three postgraduates including myself, in which I contributed significantly to the overall design of the ontology, including both the temporal structures (such as the timeline arrangement) and the spatial elements (including the use of MediaItem instances to refer to portions of media). I was also central to the implementation of extension ontologies, specifically for binding OntoMedia to film and script items, and with the development of SiX (Screenplays in XML) which was used to demonstrate these bindings.

OntoMedia, which is in use under several other projects as well as SBS, has been presented at the Semantic Web Annotations for Multimedia[3] (SWAMM) workshop, the International Semantic Web Conference[4] (ISWC), and the Multimedia Information Retrieval[5] (MMIR) workshop at SIGIR (Special Interest Group on Information Retrieval).

The mappings from the annotated media to musical parameters make use of the composer representation format. This was again created specifically for SBS, but in such a way that the configuration may be easily extended to allow for future functionality. XML was used for the file format as this may be easily parsed and provides a simple hierarchy, while individual agent configurations may use extended markup without needing to alter the basic structure. This work has been presented at the International Computer Music Conference[6] (ICMC) and the Web Delivery of Music conference[7] (WEDELMUSIC).

The Light Agent Framework (LAF) was again created as a collaboration. Based on an initial prototype by Layla Gordon, myself and Dr L. Middleton designed a standard API for the framework and implemented the result in Java, C++, and Python. This required the creation of a simple XML standard for communications, the addition of the 'ports' approach to parameter setting/retrieval, and the development of the AgentGraph structure for networked agents. I was central to the implementation of the Java version of LAF as well as these three tasks, with the resulting package used to handle the musical agents described later in this thesis. The Java and the C++ versions have both been published, with the former presented at the Knowledge-Based Intelligent Information and Engineering Systems conference[8] (KES), and the latter at both KES[9] and the International Conference on Intelligent Robots and Systems[10] (IROS) in the context of a middleware for multicamera applications.

[3] Lawrence et al. [2006]
[4] Lawrence et al. [2005]
[5] Jewell et al. [2005a]
[6] Jewell et al. [2005c]
[7] Jewell et al. [2003]
[8] Jewell et al. [2005b]
[9] Middleton et al. [2005a]
[10] Middleton et al. [2005b]

Finally, the design of the individual agents for musical composition was carried out entirely independently, with all algorithms implemented by the author (using the Java Genetic Algorithms Package[11] where appropriate) and the message passing and agent graph configuration was conceived specifically for SBS.

To summarise, aside from toolkits for algorithm design, the entirety of SBS was created either by myself or as part of a collaboration with other researchers, with the OntoMedia and Light Agent Frameworks both in use in other projects at the time of writing.

[11] http://jgap.sourceforge.net/

Chapter 2

The Traditional Composition Process

In order to create a system which could be deemed analogous to the composing process, it was first necessary to examine the techniques used by human composers to create film soundtracks, from viewing the initial movie to the final music writing stage. The movie composing task as a whole follows a process which, over time, has become a defacto standard. Kalinak [1992] sets out the key stages of film composing, and this chapter discusses these in relation to SBS.

2.1 Filming

While a movie is being shot, the director will typically use temporary ('temp' or 'scratch') music to aid with mood setting and editing. This has two key drawbacks: firstly a composer will often prefer to work with a 'blank slate' rather than simply conforming to an existing piece of music, and secondly the director may well come to identify the scene with the temporary track, and so prefer it over any other provided music. The latter case occurred with *The Exorcist* (1973) in which the entire temp score was kept, and *Platoon* (1986) where the director prefered Barber's *Adagio for Strings* over the composer's score.

However, despite these negative aspects, temp tracks are also beneficial to the composer, as they are a means for the director to convey how they believe the music should 'feel'. Directors typically have a 'role model' for their film's music, influenced by other films of the genre, and this norm may be altered to give an effect the audience does not predict, such as *Clockwork Orange*'s use of Beethoven's ninth symphony in a violent context. Furthermore, temp tracks may be advantageous when working to strict deadlines, or in

cases where the director would like to use the temporary music but is unable due to licensing or cost restrictions.

There are essentially two parts to the transfer of a director's influence to an automated composing system: the processing of existing music into a manageable form, and the incorporation of this form into the composer's 'guidelines'. The former is an active topic in music research, both from an information retrieval perspective (the transcription of digital recordings) and a perceptional perspective (locating key elements of music, such that a piece can be created in the same vein). In the case of SBS the focus is on the compositional process rather than the analytical. As such, while the system allows for the incorporation of suggestions for the style of the resultant music, techniques for the extraction of this information are considered an area for future research.

2.2 Screening

The film composer is rarely involved during the making of the film, unless music must be written before a scene can take place (such as dance music). Instead the composer views a "final cut" of the film, which is in a form close to that which the public will view. This screening process provides an initial feeling for the film.

The screening is not fully realised in the SBS system, although it is partially accommodated by the composer representation. To fully handle the procedure would require a correlation between the presented video and prior films for which the composer had written music. By comparing events matching or similar to those in the prior compositions, the composer modifications could be carried through as suggestions for the new piece of music. SBS includes technologies which would facilitate this, namely OntoMedia for the annotation and subsequent querying of media information and the composer representation to describe the composer's musical suggestions. These technologies are described in chapters 6 and 7 respectively.

2.3 Spotting

Once the film has been viewed in its entirety, the composer runs the film scene by scene to decide where music should begin and end. The producer, director, and music editor are often involved in this process, as the presence of music within the film is essential to the mood that is conveyed. Prolonging the slience before an entrance dramatically increases the emphasis on the entry, and synchronising this entry with a key line of dialogue or action heightens the effect of the film itself. The inverse is also true, with the gradual easing out of music being preferred in sensitive scenes.

The spotting process is handled using OntoMedia annotation. As is described in Chapter 5, OntoMedia allows for the description of events and entities within the film, as well as finer detail such as action events, character introductions, and plot shifts.

Karlin [2004] defines five key points where music may act as the narrator in a film, and OntoMedia is currently used to accommodate all but camera movement, which may be scope for a future extension:

1. A new emotion or subject within dialogue.
2. A new visual emphasis. For example, a change of scene, or the introduction of a new character.
3. Camera movement to support emotional emphasis.
4. A new action, such as a car driving off, or a person leaving a room.
5. A reaction to a dialogue or an occurrence (this is also an emotional response).

2.4 Laying Out

Before composition can begin, the score is laid out onto score paper. This includes the beats and bar-lines, but no notes - instead, the composer labels moments which need to be captured within the music. The composer and editor create a 'click track' consisting of clicks placed opposite the picture. This click information conveys changes in rhythm, with each 'click' corresponding to the tick of a synchronous metronome that is locked to the film. The click track is useful both to the composer and, eventually, the musicians who perform the finished score. This is the first stage which touches on the composition process, with the SBS OntoMedia annotation providing event information and the composer representation describing the beats.

2.5 Motif Development

A significant difference between film composition and concert hall composition is the length of the piece. Musical cues in film scores are typically short, primarily as the director is likely to rearrange shots and it is difficult to rearrange music when it is overly long. As such, film music often makes use of repeated short motifs, or themes associated with characters (*leitmotif*). The Harry Lime theme, as used in *The Third Man* is a pertinent example of a character leitmotif. Robertson et al. [1998] describes the work of Herrmann, whose collaborations with Hitchcock included *Psycho* (1960) and *Vertigo* (1958). Herrmann chose to focus on harmonic and rhythmic constructions, downplaying

the role of melody in his film scores, hence reducing the reliance on a coherent musical form.

Motif generation is not handled at present in SBS, although it could be made possible by producing 'seed' melodies at the start of the composition process on a per-entity basis (e.g. a seed melody for an individual character or item). Currently, composing agents may be directed towards set pitch patterns and note durations (see Chapter 8). The output from these agents can then be used to bias the composer representation to including specific rhythms and melodies.

2.6 Final Composition

Once the structure of the soundtrack has been established and motifs are selected, the final composition process begins. Here the composer utilises the styles chosen at the screening stage in combination with those from the spotting stage to build a finished piece of music. Naturally this may go through several revisions, with both the director and the composer editing where appropriate. It is at this juncture that SBS fully utilises the composing agent framework, with the composer representation and annotated media being bound together to form a collection of parameter states to influence the process.

2.7 Summary

In designing the State Based Sequencer, close attention was paid to the 'traditional' soundtrack composing workflow. From the the initial viewing of the film, to the composer's stylistic interpretations, to the eventual composition of the soundtrack, the technologies developed for SBS provide, or allow, automated approaches. Each of these technologies - the OntoMedia annotation system, the composer representation, and the Light Agent Framework - are discussed during the course of this thesis. First, however, existing techniques for the automated composition of music are considered, as these are vital to the development of the composing agents.

Chapter 3

Automating Composition

Algorithmic techniques have been used to create independent pieces of music using rule-based methods from before the advent of computing. In 1026, Guido d'Arezzo developed a formal technique for text accompaniment by assigning pitches to vowel sounds[1,2]. In the 1400s, the *golden section* was popularised, and Guillaume Dufay (1400-1474) derived the tempi for one of his motets from the proportions of a Florentine cathedral[3]. As well as this technique, Dufay also applied *inversion* and *retrograde* procedures, which respectively made positive intervals negative (hence inverting the contour of a melody) and reversed the order of pitches. These techniques were later to be used extensively in serial music.

W. A. Mozart was one of the first composers to use dice for the creation of minuets, with the publication of *Musikalisches Würfelspiel* providing sets of matrices with dice numbers delineating the rows, and the columns marked with Roman numerals indicating the portion of the piece. By locating the cell matching the dice total and part, a number was given indicating which measure should be played - effectively making it a probabilistic composition algorithm. This idea was marketed further in the 1800s, with sets of cards sold as automatic 'waltz creation kits' and composing aids.

However, while these game-based approaches made for interesting performances, they did not lend themselves to the creation of multiple musical works. This was to begin in the 20th century, as computers took on the generative task and composing algorithms progressed dramatically. This chapter examines some of the current 'stock' composition methodologies, all of which were considered as candidates for the composing tasks required in SBS.

---

[1] Kirchmeyer [1962]
[2] Loy [1989]
[3] Roads [2001]

3.1 Stochastic

The initial burst of composing algorithms began with the introduction of 'serialism'. Arnold Schoenberg founded the movement in the 1920s, beginning with the introduction of 'twelve-tone music'. To create serial music, a series of 12 different chromatic notes was selected from the 12! series available. This could then be repeated or altered via retrograde or inversion procedures, as mentioned previously. Later, Karlheinz Stockhausen adopted the phrase 'serielle Musik'[4] to distinguish his music from this twelve-tone music technique and, together with Pierre Boulez, John Cage, and Earle Brown, he experimented with *aleatoric* composition. As with Mozart's approach this left details of interpretation to the performer, with a random technique used to select the direction of the piece, and this 'handing over' of performance direction to random parameters became a foundation of serial music. A pertinent example of serial music is that of *Structures 1a* by Boulez. In this piece, a set of magic squares are provided, with the rows and columns providing the pitch and rhythm while dynamics and attack are given by the diagonals. For example *pppp* (very quiet) is assigned to 1, and *ffff* (very loud) is assigned to 12. Serialism was, as a result of its probabilistic nature, ideally suited for computational adaptation.

For a more flexible approach to composition, Markov chains are often employed. These are discrete probability systems in which the probability of future events depends on one or more past events[5]. Markov chains have an associated order, which specifies the number of previous events which are considered at each stage, hence an Nth-order Markov chain can be represented by an N+1-dimensional probability table as it provides the probability of an event given the previous N states. This is ideal for cases such as harmony generation, where the next chord may rely on the prior two chords. Furthermore, cycles may be easily defined using Markov models, as they are simply a previous event leading to the same event. This is useful for rhythm selection, where a short phrase may be repeated frequently.

Markov Models require an initial training process, either from an existing piece of music[6] or from parameters. For example, a melody could be converted into a 1st order Markov Model, with each element in a 2D matrix containing the probability of moving from one note to another. This process has been used to produce very aesthetically pleasing music[7], but the extensive training requirements and the inability to synthesise beyond the initial configuration are significant hindrances. Markov models are often used in combination with other composition methods, such as a means to create probabilistic grammars or to control the chromosomal content in the fitness function of a Genetic Algorithm.

---

[4] Grant [2001]
[5] Miranda [2001]
[6] Jones [1981]
[7] Farbood and Schoner [2001]

3.2 Cellular Automata

Cellular Automata (CA) were first described by Ulam and Von Neumann and were the combination of Ulam's research into crystal growth and Von Neumann's research into self-replicating systems. A CA is implemented as a collection of 'cells', typically arranged in a lattice, which individually obey a defined set of behavioural rules. As time progresses each cell alters depending on three factors: its previous state, the states of the cells in its immediate neighbourhood, and the transition rules for all of the cells in the system. Wolfram[8] states that the 256 "elementary" CAs (2 states and examining one cell in the neighborhood) can be classed by one of four behaviours:

1. The system collapses as the cells consume the random structure
2. Evolution leads to fixed or pulsating periodic behaviour
3. Chaotic patterns ensue in the lattice, containing instances of organized behaviour
4. Strange attractors emerge, with unpredictable behaviour appearing as a result

John Conway's 'Game Of Life' arrangement is the most prominent form of CA, taking a localized state approach. Conway's technique consists of a 5-cell automaton - a central cell surrounded by neighbours at its north, east, west, and southerly points. Four rules are set in place:

1. Birth: A cell that is dead at time t becomes alive at time $t + 1$ if exactly three of its neighbours are alive at time t
2. Death by overcrowding: A cell that is alive at time t will die at time $t + 1$ if four or more of its neighbours are alive at time t
3. Dead by exposure: A cell that is alive at time t will die at time $t + 1$ if it has zero or one live neighbours at time t
4. Survival: A cell that is alive at time t will remain alive at time $t + 1$ only if it has either two or three live neighbours at time t

Two values may be used to parameterize the evolution of a Game of Life CA, namely the environment (E) and the fertility (F), where E is defined as the number of living cells surrounding a live cell, and F the number of living cells surrounding a dead cell. In the rules above, survival occurs when $2 \leq E \leq 3$ and birth occurs when $3 \leq F \leq 3$. The standard Game of Life model is therefore denoted as 23/3: the cell survives if there are 2 or 3 neighbours, and a new cell is born if there are 3. Figure 3.1 shows a small Game Of Life neighbourhood at time t, $t + 1$, and $t + 2$.

[8] Wolfram [1984]

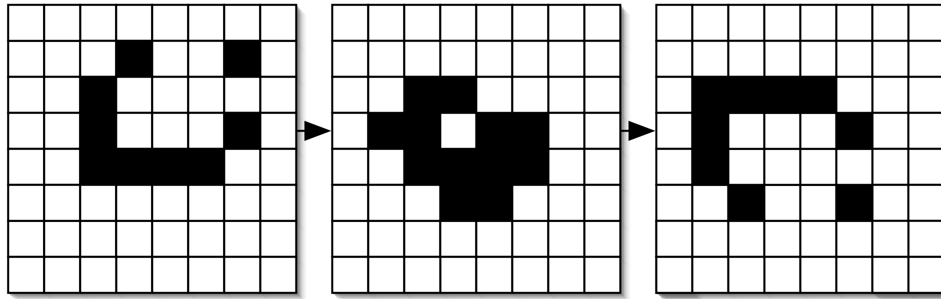


FIGURE 3.1: Three stages of a Game of Life cellular automata. This configuration is known as a 'lightweight spaceship' (LWSS) which progresses along the lattice over subsequent iterations.

3.2.1 Demon Cyclic Space

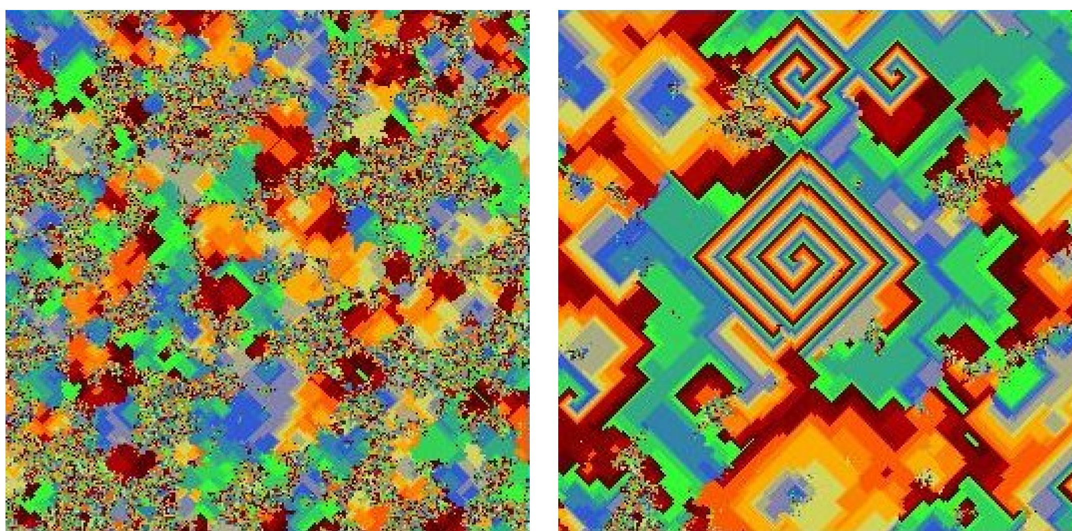
A further, more compositionally-inclined, variant of a cellular automata is the Demon Cyclic Space. Compared to Game Of Life, DCS operates on all adjacent cells and, rather than simply 'dead' and 'alive' states, assigns a value from 0 to $n-1$. Furthermore, DCS wraps around the edges of the grid, forming a torus. It follows two rules:

1. A cell which is in state k at time t dominates any adjacent cells which are in state $k-1$ at time $t+1$ (i.e. these cells change from $k-1$ to k)
2. A cell which is in state $n-1$ at time t may only be dominated by a cell which is in state 0 (hence the cyclic attribute of this automata)

Iterating a Demon Cyclic Space from an initial random lattice produces a stable pattern of spirals, with each spiral classed as a 'demon'. Figure 3.2 shows an example of this, with 3.2(a) illustrating the initial 'domination' stage, and 3.2(b) the eventual spiral pattern.

3.2.2 Composing with Cellular Automata

Initial experimentation into CA-composed music was instigated by Beyls in 1989, making use of what Wolfram described as the "large numbers of simple identical components with local interactions". This research considered a one-dimensional automata - essentially a 'strip' of cells as opposed to the more common lattice. The traditional Game Of Life, combined with Demon Cyclic Space, are now useful tools for the composition of algorithmic music. These techniques are most prominent in the Cellular Automata Music (CAMUS) software.



(a) The domination stage of a Demon Cyclic Space.

(b) The spiral 'demons' produced after further generations.

FIGURE 3.2: Two figures showing the development of a Demon Cyclic Space. Beginning with a random configuration, (a) shows the initial signs of domination, while (b) shows the development of stable spiral patterns.

3.2.2.1 CAMUS

The CAMUS 2D system[9] applies the ideas behind CA to musical composition, employing both the Game of Life and the Demon Cyclic Space. At time t, the coordinates of live cells in a multi-level Game of Life are analyzed and converted into a three-note chord. First, a fundamental pitch is selected from a predefined sequence specified by the user. The horizontal position of an active cell gives the interval between this pitch and the bottom note, while the vertical gives the interval between it and the top note. For example, if cell (6,5) is active and the fundamental is C, the bottom note will be F# and the top note will be F.

The Demon Cyclic Space is then used to determine the instrumentation of the chord. The state of the corresponding cell in DCS provides the instrument number (MIDI Channel). Finally, information from the neighbouring cells in the Game Of Life provides temporal variation. a, b, c, and d are set to 1 if the southern, northern, western, and eastern neighbours respectively are active, and m, n, o, and p are set to 1 if the south-west, north-east, south-east, and north-west cells are active. Next, a bitwise inclusive OR operation forms two four-bit words: $Tgg = abcd|dcba$ and $Dur = mnop|ponm$. These represent the note trigger information (Tgg) and duration (Dur) and are assigned via a look-up table approach.

[9] Miranda [2003]

CAMUS 3D, also by Miranda, adds an extra z co-ordinate. This provides a four-note grouping instead of the original triad, and also makes use of a first-order Markov Chain for temporal coding instead of the neighbour analysis of the 2D approach.

3.3 Grammatical Production

The grammar-based approach to algorithmic composition is rooted in the study of linguistics, which designates the formal system of principles or rules by which the possible sentences of a language are generated (Burns). A grammar is defined by the four tuple $G = (N, \sigma, P, S)$, where N is a set of nonterminals (i.e. symbols that may be replaced), σ is a disjoint set containing terminal symbols (i.e. symbols that may not be replaced), P is a set of production rules, and S is the 'start symbol' and hence must be present in N.

Production rules govern how the initial symbol is transformed throughout the iterative process. These provide a left-hand string, containing at least one nonterminal, and a right-hand which may be entirely terminals if desired. For example, given the production rules $S \rightarrow aSb$ and $S \rightarrow ab$, the start symbol S may initially be transformed into either aSb or ab. Subsequently, if rule 1 was selected (and hence a nonterminal remained) the string could progress to aaSbb or complete at aaabbb.

Grammars may be restricted further by switching from context-free grammars (such as shown previously) to context-sensitive. These allow terminals on the left-hand side of the production rules, hence only allowing production to occur if all symbols are in the correct locations. For example, $aSb \rightarrow aaSbb$ is an example of a context-sensitive production rule, as S must be between a and b for the production to occur.

3.3.1 Composing with Grammatical Productions

Early studies into the use of grammars for musical composition took place in the late 70s, with Buxton et al. [1978] and Roads [1979] providing initial groundwork.

L-Systems, as used by McCormack [1996], lend themselves well to musical composition. Named after Arstrid Lindenmayer, a Hungarian theoretical biologist, L-Systems are very similar to traditional grammars (i.e. a set of terminals, nonterminals, production rules, and an initial axiom). L-Systems allow for a string as the initial axiom, however, from which the production rules are applied iteratively. L-Systems are particularly well suited for biological forms, as the recursive nature is ideal for the representation of neighbourhood relationships and branching structures.

To apply L-Systems to music, McCormack generates a string with symbols instructing the player system which actions should be performed. '.' results in the current note

being played, '+' increments the pitch by a semitone, '-' decrements by a semitone, upper-case letters play a note (e.g. C), and lower-case notes are stored to be played by a future '.'. This system also provides for polyphony, with notes in parenthesis played simultaneously. For example, (CEG) is a C major chord, as is c(.+++.+++.). Context sensitivity is provided through the use of both polyphonic and temporal contexts, with $(CE)|(GC) \rightarrow D$ indicating that D should be played if the current chord is (GC) which is immediately preceded by (CE). This flexible production rule syntax allows for complex musical generation, with the option of parameterization using parametric grammars.

3.4 Fractal

While the term 'fractal' was coined by Benoit Mandelbrot in the 1960s, the idea of fractal behaviour was developed much earlier, and noted in the early 1500s regarding fractal geometry in craftwork. Leibniz first developed the idea of recursive self-similarity, and this concept was then formalized into a function by Karl Weierstrass which Helge von Koch defined in a geometric definition[10]. Mandelbrot defines a fractal as being a set with Hausdorff-Besicovitch dimension D_n strictly exceeding the topological dimension D_t. Geometrically, fractals are similar at all scales of magnification.

3.4.1 Composing with Fractals

In 1975, Voss and Clarke[11] determined that music, once high-frequencies were eliminated, resembled $1/f$ noise. This was a provocative suggestion, partly due to music being highly organized, with deterministic leanings, and partly that they suggested this applied to all music, which is a very general statement. Varieties in instrumentation, performance style, and even acoustic conditions affect recorded music. Beran [2004] determines that not all compositions resemble $1/f$ noise, however almost all compositions resemble $1/f^\alpha$ for some $\alpha > 0$, and deviations are rare. α can be related to the fractal dimension D, as $D = (5 - \alpha)/2$.

Voss and Clarke took this research further and applied the principle to musical composition, using a parameterisable noise generator as a pitch selector. They found testing with white, uncorrelated, noise, produced correspondingly random notes, whilst Brownian ($1/f^2$) gave overcorrelated melodies. Using $1/f$ noise, however, gave natural sounding melody.

Leach and Fitch [1995] later employed this characteristic, with peaks in a non-linear system denoting 'major notes' - effectively the pivots in the music. This gave contours, with sequences directed towards major notes, and the rhythm and melody were only

[10] von Koch [1904]
[11] Voss and Clarke [1975]

permitted to change at these points. This is comparable to representing the music as energy, with the potential energy highest at major notes.

3.5 Genetic Algorithms

Based on the framework created by Holland[12] in 1975, Genetic Algorithms are modeled on Darwinian evolution. This considers a population of individuals, from which a sample is selected based on a 'fitness function' - essentially survival of the fittest. Each solution consists of a collection of chromosomes, which are further made up of numerous 'allele'. These are the smallest structures in the system, and hence are often used to represent notes when applied to music.

At each generation, a variety of operators can be applied to the population with crossover and mutation being the two most common. Crossover takes a segment from the chromosome and switches it with a segment from another chromosome. This may be a single segment chosen using a single locus (single-point crossover) or several segments chosen using several loci (multi-point crossover). Mutation selects a random allele from a chromosome and mutates it in a defined manner. For example, this could involve a number being set to a random value, or being raised or lowered. For the purposes of music generation, specialist operators are often employed, such as transposition (raising and lowering notes), inversion (subtracting the note value from the octave), and retrograde (reversing a section). As mentioned previously, the latter two operators were both fundamental to the serialist method of composition.

To further customise the evolution of the system, the process of selecting chromosomes for the next genotype can be modified. This can vary from random selection (both parents chosen completely randomly) to fitness-based approaches. Tournament selection chooses a number of chromosomes from a population and selects the fittest of these. Alternatively, roulette wheel selection uses the fitness level to associate a probability of selection with each individual chromosomes - effectively giving each chromosome a segment on a wheel, with its size dependent on the fitness.

3.5.1 Composing with Genetic Algorithms

The initial incursion of genetic algorithms into the stock collection of automated composing techniques began in 1991, with Horner's application of GAs to thematic transformation[13]. Biles[14] described GenJam, a genetic approach to jazz solo generation. This made use of human feedback from a 'mentor' who, while listening to a solo, typed 'g's

[12] Holland [1975]
[13] Horner and Goldberg [1991]
[14] Biles [1994]

if the portion was judged to be good, or 'b's if it was judged to be bad. This modified the fitness value of the candidate solo, and hence the population altered accordingly. GenJam used two separate populations in its GA, one for measures and one for phrases, with the genes of the former mapping to a sequence of MIDI events and the genes of the latter mapping to individuals in the measure population. Biles further applied 12 mutation operators, 6 specifically for measures and 6 specifically for phrases, with the measure operators including the traditional retrograde, invert, transpose, and rotation. The phrase operators were less conventional, with a 'lick thinner' ensuring phrases did not occur too often and a 'super phrase' generating an entirely new phrase by selecting the indices of the winners of four independent three-measure tournaments. The area was further progressed by Horowitz[15], who generated rhythm using GAs, Jacob[16], who used GAs to produce filters for a stochastic musical generator, and Thywissen[17], who developed an interface for the evolution of music using grammars.

Wiggins et al. [1999] employs a penalty-based approach in his alternative technique for jazz melody generation. The fitness function has a collection of weightings which can be applied to the final overall fitness. For example large intervals, non-scale notes at downbeats, and long dissonant suspensions all incur penalties, while chord notes at downbeats and consonant suspended notes are weighted favourably. This approach of weighting based on musical factors is utilised in the the melody and rhythm agents of SBS, a fact which is discussed later.

3.6 Summary

The State-Based Sequencer uses a Genetic Algorithm approach at the core of the majority of its agents. In the case of several of these (Key, Chord, and Melody being pertinent examples) a Markov Model is employed at the fitness function level to allow easy parameterisation. Genetic Algorithms provide a means to generate a result which may not be what the user is expecting, simply by altering the threshold at which the eventual result is chosen. Furthermore, as was described above, it is possible to train a Markov Model using existing pieces, which would allow for the composition of music based on the parameters of an existing composer. This flexibility was ideal for the project. The two agents which do not use GAs (Tempo and Instrumentation) instead use simple functional approaches to produce their results, which are described in Chapter 8.

[15] Horowitz [1994]
[16] Jacob [1995]
[17] Thywissen [1996]

Chapter 4

Media Annotation

4.1 Describing Media

The annotation of media into an ontological form has been investigated from a variety of angles by prior researchers, most significantly by Lagoze and Hunter's ABC Ontology[1]. This was designed primarily for the cataloguing community, with a focus on factual information representation, such as object provenance and rights management. ABC and OntoMedia share the separation of entity and temporal classes, while the OntoMedia ontology augments this core with classes that allow for specialization to other contexts. Furthermore, Hunter proposes a technique to represent MPEG-7 using a DAML+OIL representation, whereas OntoMedia references segments of the source media using a customisation of VLit location specifiers.

The cultural significance of annotation ontologies is further emphasised by the CIDOC Conceptual Reference Model (CRM), which was created as a "semantic approach to integrated access"[2] for cultural heritage data. By providing a conceptual basis that can be used for automated mapping the CIDOC CRM acts as a bridging technology between existing data structures, and a guide to creating new structures.

Once a framework for the description of multimedia has been chosen, it is then necessary to annotate the chosen media. While this may be done manually, it is more useful to provide a means to automatically locate and annotate pertinent areas of the item. Full automation is a very difficult task - subtle changes in a scene may be of significant importance to the plotline, or cast members may intentionally hide their appearance from the viewer - so at present it is more convenient to provide tools to augment manual markup. As SBS works with film there is a wide range of monomedia, including the video itself, sound effects, speech, diegetic music, closed caption, and script information.

[1] Hunter [2003]
[2] Crofts et al. [2005]

While automatic annotation is not at the core of the SBS project several techniques were considered, and some utilised to ease the annotation process.

4.2 Video Analysis

Video is the richest information source available from film. At a high level it is possible to examine colour, movement, and transitions, while at a lower level more complex data can be retrieved, such as character recognition.

4.2.1 Colour Detection

The colour within scenes provides an immediate suggestion as to the mood. In western cinema, red is often used to suggest a sensual environment, whereas white is taken to represent innocence. David Lynch has a directorial trademark of using red curtains to suggest a dreamlike quality (see Figure 4.1). In Roman Polanski's *Tess*, he uses red and white to suggest the peasant heroine's transition between sin and innocence, with Rabiger [1997] highlighting the "white dresses in the opening May walk [and] the red of the strawberry Alec puts between her unwilling lips" as pertinent examples. Some directors make use of a sparse use of colour to highlight scenes: Steven Spielberg filmed Schindler's List in black and white, except for one scene where a girl's red dress was shown in colour - the effect is later used to show the dress amongst a pile of bodies.

FIGURE 4.1: The use of the colour red to represent a dream in David Lynch's *Twin Peaks*.

Some films are entirely centred around a single colour scheme, such as Krzysztof Kieślowski's Three Colours Trilogy. Each of the three films focuses on a colour from the French flag, together with the associated ideal of the French Republic's motto (liberty, equality, fraternity). As such, Three Colours: Blue uses the colour blue heavily, with blue filters, blue lighting, and blue objects representing the lead character's past; Three Colours: White contains many white scenes to suggest hope, innocence, or bleakness; and

Three Colours: Red indicates love with its signature colour, whether on automobiles or advertising banners. Furthermore, Kieślowski also brings some of the other two colours into each film. One scene in Blue features a blue swimming pool, with children dressed in white bathing suits and red armbands.

Detecting the colour within the scenes requires a histogram technique that considers the spatial configuration of pixels with the same colour[3], so either the CCV-based Pass and Zabih[4] or the augmented Chen and Wong[5] technique would be appropriate. Unfortunately it is very difficult to judge color, as so many shades and combinations are available, and callibration becomes an issue. It is feasible, however, to locate very strong colours (such as in the David Lynch scene pictured) or areas where an area of colour is strongly contrasting with the rest of the scene. A useful approach is to transform the colour space to a representation that is illumination invariant, such as HSV.

4.2.2 Motion Detection

Whilst it is very difficult to determine the 3D motion of the objects in an arbitrary scene, 2D information is more readily obtainable and can be used to good effect. Examining key features can provide information on camera pan, tilt, and zoom, which can then influence the music. *The Matrix* made use of zooming to accentuate a scene change to a building top, with fast descending scales as the camera zoomed into the top of the building, and ascending scales as the camera zoomed out (see Figure 4.2).

FIGURE 4.2: Two characters falling onto a building in *The Matrix*.

Ewerth et al. [2004] proposes a three-step approach to camera motion estimation for MPEG. Motion vectors (MVs) are first extracted from the P-frames, which contain information relative to the prior frame. Next, a noise removal process is applied: opposing neighbours to the motion vector are averaged, with the MV being removed if it is not close to a significant number, and MVs are removed if a specified number of neighbours

[3] Wang et al. [2000]
[4] Zabih et al. [1999]
[5] Chen and Wong [1999]

do not lie within the same tolerance circle as the candidate MV. Finally, the technique defined by Srinivasan et al. [1997] is employed, which combines flow information with a 3D camera model to extra camera parameters.

Further details can be obtained from the speed of the motion and the level of excitement within the segment (i.e. the proportion of pixels that are moving in different directions). A composer may choose to increase the tempo when this level increases - a technique that is ideal for action movies, although these also rely on fast scene changes to build momentum.

4.2.3 Transition Detection

The motion information stated previously can also provide a means to determine the location of shot transitions in a film. Zabih et al. [1999] describes a technique that uses intensity edge comparison between frames to detect and classify production effects including cuts, fades, dissolves, wipes, and captions. Their method uses an edge change fraction, which is the maximum of the proportion of entering and exiting edge pixels. Peaks in this fraction denote scene breaks within the sequence.

The Zabih technique was applied to a segment of *The Matrix*, with the edge change fraction computed by applying a Canny edge detector, duplicating the image and applying a diamond dilation (i.e. replacing each edge pixel with a diamond of a specified radius), and then counting a black pixel as an exiting pixel when the corresponding pixel in the original edge image was not black. The results from this can be seen in Figure 4.3.

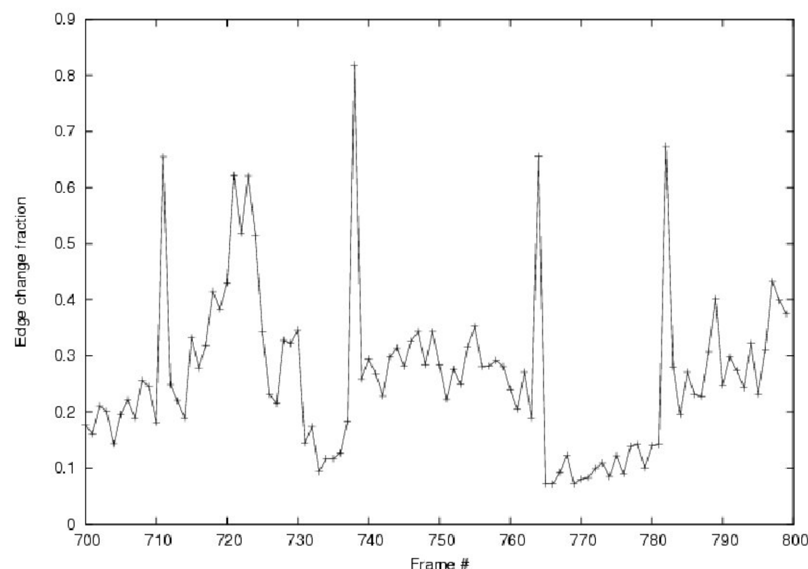


FIGURE 4.3: The edge change information computed for a segment from *The Matrix*. The peaks represent shot transitions present within the sequence.

4.2.4 Cast Member Identification

While the scene location information can be used in combination with a script to locate cast members, ideally it should be possible to remove the need of a script entirely. Fitzgibbon and Zisserman [2002] describes a technique using an invariant distance function that is able to extract the principal cast members from a movie sequence. This uses the Schneiderman and Kanade face detector, with their implementation obtaining a positive rate of about 80% of frontal faces, and then clusters filtered versions of these faces incorporating deformation and speed priors. Fitzgibbon's process produces very good results, although some duplicates are detected due to facial expressions.

4.3 Audio Analysis

4.3.1 Vocal Detection

One of the harder problems in generating a soundtrack is knowing where *not* to place music. For this reason, locating characters' speech is essential to ensure no dialogue is obscured. Yuan-Yuan et al. [2004] proposes a feature-based approach to discriminating between vocal and environmental sounds, with 9 features calculated from pitch contours evaluated by a neural network. This approach has a 98.73% hit rate with an 11% false alarm rate, which is suitable for an estimation of the location of vocal segments.

4.3.2 Music Detection

While we assume that no external music is present during annotation, it is possible that diegetic music (i.e. music which is part of a scene) will be used. For SBS, this is an indicator that no non-diegetic music should be provided for that section, but it may be useful to do further analysis and recognise the piece of music in the segment. Minami et al. [1998] describe a technique utilising edge detection on a sound spectrogram for this purpose. The power spectrum is calculated using an FFT, representing it as a grey-scale image which is then convoluted by an edge detection operator. When music is present, the spectrum peaks tend to settle at certain frequences, and as such the total edge intensity becomes high. Using a threshold it is therefore possible to detect speech and music segments.

4.3.3 Foley Detection

Foley sounds, named after Jack Foley (a sound engineer for Universal Studios) are artificially produced sounds imitating natural noises. If the previous two detection stages

are accurate the location of Foley is less difficult, but classifying the sounds is much harder. Wold et al. [1996] analyses several features in the sound, including loudness, pitch, brightness, bandwidth, and harmonicity, and creates a feature vector per sound. To classify sounds, a distance measure is classified from the new sound's N-vector and Euclidian distance is used. Furthermore, if some features are known to be unimportant these can be disregarded or given a low weighting. By analysing segments of the remaining audio track, it would therefore be possible to classify Foley sound.

4.4 Script Annotation

To ease the annotation of screenplays into a machine-readable form, a simple format was designed. Named SiX (Screenplays in XML), the format is similar to HTML in that it wraps around existing text. So, given a script, very little alteration is required to render it readable. Four 'core' tags are provided: transition, location, dialogue, and direction. Transition tags denote a change in scene, and can be cuts, fades in, fades out, dissolves, or blackouts. Locations are also straightforward, with two attributes ('time' and 'pos') specifying the time period (day or night) and the position (interior or exterior) respectively, and the enclosed text giving a more detailed description.

The dialogue tag, as expected, denotes speech within the script. It has three attributes: 'paren' and 'speaker'. Paren, short for parenthetical, allows for a description of the manner in which dialogue should be spoken (e.g. 'nervously'), while speaker identifies the character delivering the dialogue. An optional 'voiceover' flag is also allowed, which specifies that the text is read by a narrator. Finally, the direction tag, which does not take any attributes, simply describes an action taking place in the script.

To provide a suitable amount of metadata relating to the script, SiX allows for Dublin Core annotation inside an <sc:info> block. As such, it is possible to denote the creators, creation date, a description of the script, and (most importantly) the title of the work. To ease readability of SiX documents, a custom XSL was created to render them into a style conforming to the Oscar requirements for submitted screenplays[6]. This includes margins, font sizes, and case requirements. Listing 4.1 shows a portion of *Apocalypse Now* annotated using SiX.

Every element in the SiX format may also contain an optional 'id' attribute. This is primarily for the reference of SiX elements from OntoMedia, described later. There is also the possibility of adding unique identifiers for speakers with the 'speakerid' attribute, as this id is likely to be reused in the script, where other ids should be unique to the SiX element.

[6]http://www.oscars.org/nicholl/format_a.txt

```
<sc:script>
  <sc:transition type="dissolve" />
  <sc:location time="day" pos="ext">A Street in Saigon</sc:location>
  <sc:direction>
  A Saigon boom street in late 1968. There are bars and shops for servicemen;
  the rickshaws, the motorbikes.  Our VIEW MOVES TOWARD one particular
  officer; B.L.  WILLARD , in uniform, a Captain of the Airborne, followed
  by four or five Vietnamese kids trying to shine his shoes and sell him
  things.
  </sc:direction>
  <sc:dialogue speaker="Willard" voiceover="true">But I know how it started
  for me -- I was on R. and R.  in Saigon; my first time south of the DMZ in
  three months. I wasn't sure, but I thought this guy was following me.
  </sc:dialogue>
  <sc:direction>Willard looks back.</sc:direction>
</sc:script>
```

Listing 4.1: An annotated excerpt from Apocalypse Now

4.5 Summary

As was stated earlier, SBS is not currently focused on automatic annotation. However, a number of the techniques mentioned have been used to good effect. Transition detection has been implemented in a simple form, and is capable of locating shot transitions. This is useful for marking segment and beat locations in the media. Secondly, the SiX script annotation is in use and tied to OntoMedia, as is demonstrated in the case study in Chapter 9. There is future scope for the adoption of other techniques, but ideally these should be incorporated into the film creation process itself. For example, the script writing process could produce SiX-formatted XML as a byproduct.

Chapter 5

The OntoMedia Ontology

5.1 A Brief Introduction to Ontologies

The World Wide Web consists of a vast amount of heterogenous information stored in HTML as well as other media such as photographs, video, and audio. HTML is able to describe a document's structure and layout, but is limited in its ability to further classify information within a page. As such, information retrieval is hindered and much useful knowledge is unobtainable. The Semantic Web, designed to be the next evolution of the World Wide Web, aims to describe web resources with context independent, machine processable, standards.

At a low level, the Semantic Web is built on a collection of URIs (univeral resource indicators) in a triple-based structure, with each triple typically having a subject (the resource being described), a predicate (the trait or aspect of the subject), and an object (the object of the relationship or the value of a trait). This is known as the N-Triples form of RDF (Resource Description Framework). At a higher level, RDF/XML is often used as it is simpler for human readers to understand and can still be translated into N-Triples for applications to process.

To provide the capabilities to describe the knowledge of a particular domain, RDF is further extended by OWL, the Web Ontology Language[1]. This has three variants (Lite, DL, and Full) with increasing levels of expressivity, with OWL Lite and DL designed such that any statement can be decided in finite time while OWL Full may loop endlessly. At its least expressive, OWL provides features for classes, properties, restrictions, restricted cardinality (0 or 1), equality checking (both between classes and between individuals), and versioning. For example, it may be defined that a CD class is different from a Movie class, but has an equivalent property 'artist'. At its most expressive, functionality for

[1] http://www.w3.org/TR/owl-features/

```
select Painter, Painting, Technique
from   {Painter} rdf:type {cult:Painter};
  cult:paints {Painting} cult:technique {Technique}
using namespace
  cult = <http://www.icom.com/schema.rdf#>

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?mbox
WHERE
  (?x foaf:name "Johnny Lee Outlaw")
  (?x foaf:mbox ?mbox)

SELECT ?family , ?given
WHERE
  (?vcard vcard:FN "John Smith")
  (?vcard vcard:N ?name)
  (?name vcard:Family ?family)
  (?name vcard:Given ?given)
USING
  vcard FOR <http://www.w3.org/2001/vcard-rdf/3.0#>
```

LISTING 5.1: Example queries in SeRQL, SPARQL, and RDQL respectively.

Boolean combinations (i.e. unions, complements, and intersections) and more descriptive minimum/maximum cardinality is available.

Once an item has been annotated using an ontology, it is then necessary to allow for interaction with the marked information. For this, a triple-store is used. As suggested by the name, triple-stores store the RDF in an N-Triples style (subject, predicate, and object) and then provide facilities for information display and querying. For the purposes of this application, Sesame was used due to its Java API and its straightfoward maintenance. Sesame allows for querying using SeRQL (Sesame RDF Query Language), as well as SPARQL (Simple Protocol and RDF Query Language) and RDQL (RDF Data Query Language), which vary in syntax and influence but share much of the same functionality (see Listing 5.1).

5.2 The Structure of OntoMedia

OntoMedia was developed with the intention of being able to represent a heterogenous set of multimedia[2][3][4], including film, speech, fiction, and historical records. As such, the design needed specific structures to encapsulate both temporal and non-temporal concepts. At a base level it provides the facilities to annotate the fundamental content of a media item: the items and people involved, the events which occur, and both abstract and physical attributes of the entities. This base level is then extended to provide a finer level of granularity, such as gender, mood, and species.

[2] Lawrence et al. [2006]
[3] Lawrence et al. [2005]
[4] Jewell et al. [2005a]

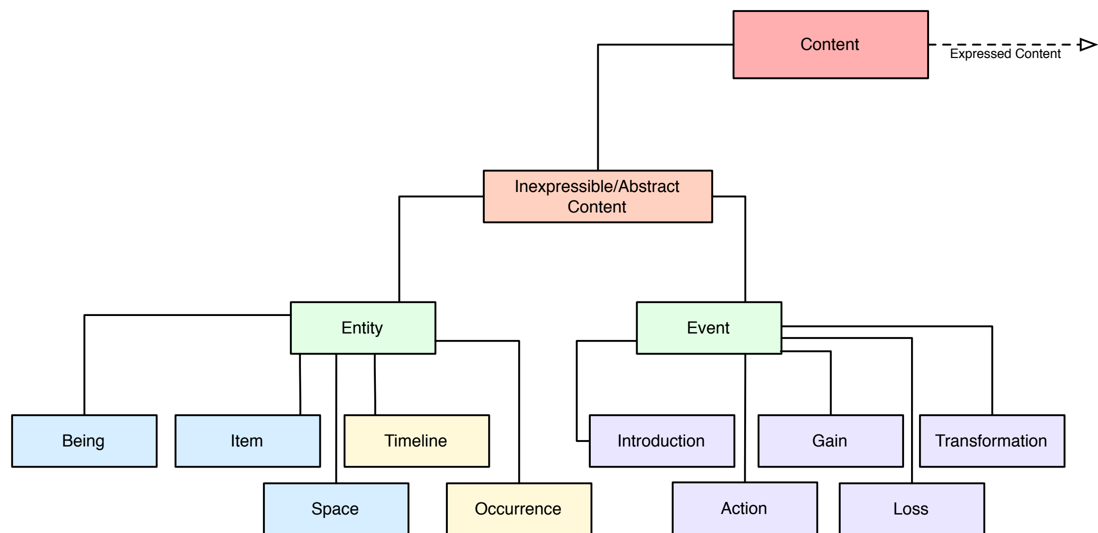


FIGURE 5.1: The class structure of the OntoMedia ontology

OntoMedia is split into three sections, at the centre of which is the 'core' ontology, which represents the domain knowledge of the system. This provides the minimum set of classes required to represent a temporal medium, and can be utilised for high-level inference even if the extending ontologies vary. For example, a fictional item may use different classes to a factual item, but they will share the 'core' classes and can hence be compared at that level.

Of secondary importance are the 'extension' classes. These directly subclass the core classes, providing resources for the annotation of different types of media. At present, this includes a 'fiction' extension, which includes characters, species, and a taxonomy of attributes useful for the description of fiction, and a 'fact' extension, which instead of 'character' provides a 'being' class to represent humans and animals. Extension domains do not have to be independent - the fiction classes make use of the 'being' class from fact, with a character played by an instance of a being.

Finally, a tertiary set of classes - the 'miscellanous set' - is provided. These classes do not extend the core or extension classes, instead providing separate ontologies which may be used in other areas. For example, a Geometry ontology is defined to represent polygons, circles, squares, and other shapes - essential for purposes such as tying actors and objects to regions in a movie.

5.3 The Ontology

As with ABC, OntoMedia splits its classes into two categories: spatial and temporal. The former is reponsible for items or concepts within a media, and the latter places

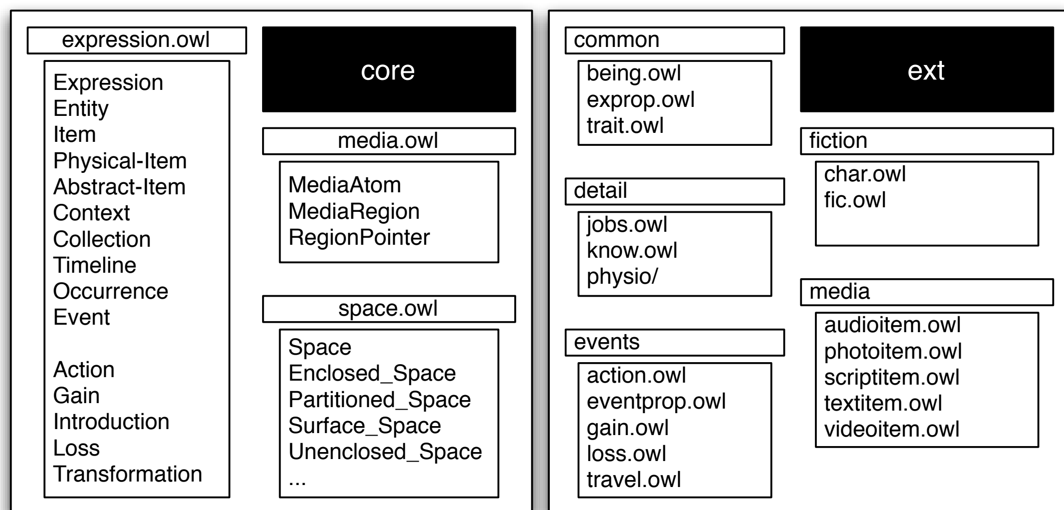


Figure 5.2: The core and extension modules within OntoMedia, with the classes of the core modules listed.

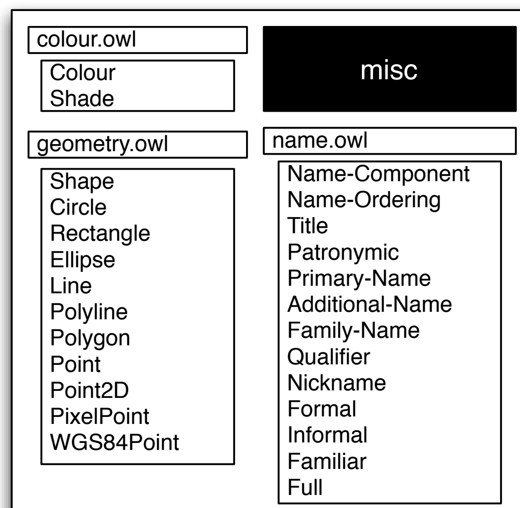


Figure 5.3: The OntoMedia miscellaneous modules. These do not rely on the core or extension classes.

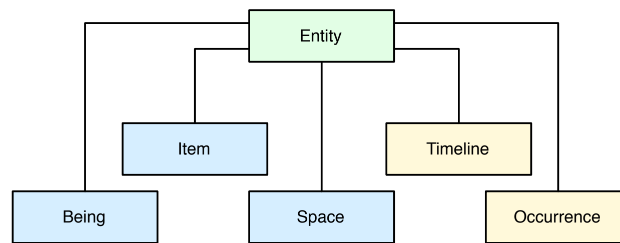


FIGURE 5.4: The OntoMedia entity model

these objects into the time domain. Figure 5.1 shows the structure of these classes, with the spatial classes on the left of the diagram and the temporal on the right, and Figures 5.2 and 5.3 show the modules of the system.

5.3.1 Entity Modelling

The spatial classes defined by OntoMedia all stem from the Entity class, as shown in Figure 5.4. Five classes subclass this to provide Being (people and animals), Item (props and abstract items), Space (such as locations), Timeline (a sequence of Occurrences), and Ocurrence itself (an instance of an Event). The Entity class provides a few key properties which are inherited by both the abstract and physical subclasses. These include container information, allowing for one entity to be contained by another, location information, which refers to a custom location ontology, and a collection of 'traits'. These are subclasses of abstract item and are fundamental to the OntoMedia representation, as they embody the characteristics and properties of entities within the media.

Several traits are predefined within the OntoMedia extension classes, including: personal information, such as age and gender; physical information, such as build and distinguishing marks; and state-based attributes, such as being and form. Through the use of a Transformation event (described in the following section), the values of these properties may be altered as the narrative progresses (e.g. adding distinguishing marks, increasing age, or the death of a character).

A powerful addition to the OntoMedia trait collection is that of motivation. By specifying a collection of entities and events, it is possible to define a state which the character wishes the narrative to achieve. For example, a character may aim to have gained a specific item and be in a certain location by the end of the media. SBS, as well as other applications, may make use of this to powerful effect. For example, the music will be likely to change style when a character achieves a motivation, and even more likely to change if all of a character's motivations are fulfilled.

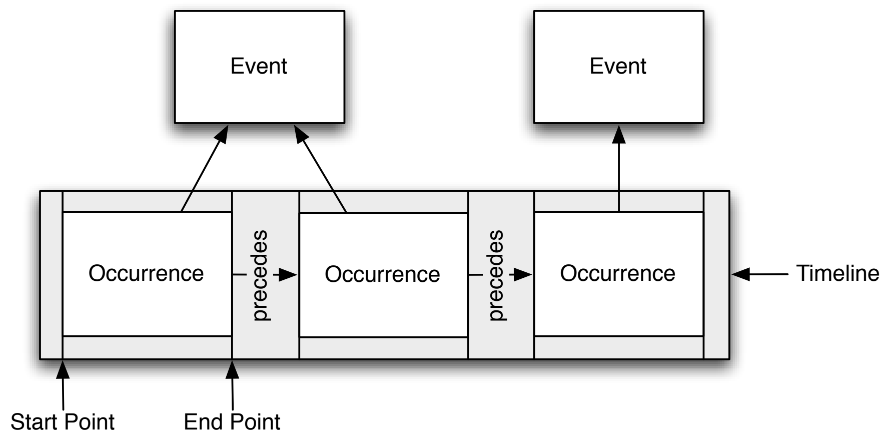


FIGURE 5.5: The OntoMedia Timeline structure

5.3.2 Event Modelling

Three core classes are provided from temporal representation in OntoMedia: Timeline, Event, and Occurrence (see Figure 5.6). The addition of a timeline is a novel approach to representation, primarily as prior models did not need to denote the passage of time. Multiple timelines may be defined, stretching from one time to another, and each timeline contains multiple Occurrences of Events. The multiple timeline approach was justified, as it allows for the annotation of dream sequences, viewpoints (c.f. *Death of a Salesman*), and storyline effects such as montages and split screens. Moreso, it is not essential for a timeline to actually occur - it may be used to specify the character motivations as described previously.

The Occurrences which reside within a timeline contain an instance of an Event (allowing for events to occur multiple times), a start and end point, and a property denoting which Occurrence they precede. This allows for the temporal relationship between Occurrences to be represented. In effect, Occurrences are simply a range of time during which an Event takes place (see Figure 5.5).

Events, in contrast, are more complex - although not overly so. They are defined as 'an interaction between one or more entities during which zero or more traits of those entities are modified and/or a new entity is created' and carry a Location (the place where the Event occurs), a set of participant Entities, and a textual description for visualization purposes. The power of the OntoMedia Event model is supplied by the provided hierarchy, which closely follows Bal [1997] and Chatman [1978]'s representations of the key aspects of narrative writing. As such, the base Event is subclassed by Gain, Loss, Transformation, and Introduction. Gain is an event in which the participant entities gain attributes, such as a character gaining an object or becoming angry (effectively gaining the 'angry' trait). Loss is the opposite, denoting when entities lose attributes.

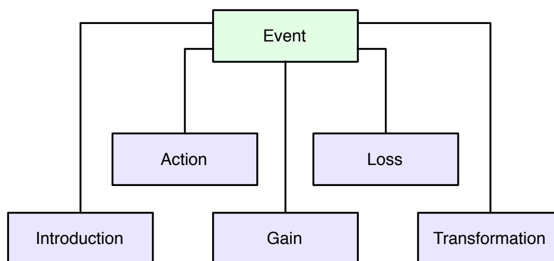


Figure 5.6: The OntoMedia event model

As such, if one character (A) gives another character (B) an item, both a Gain event (B gaining the item) and a Loss event (A losing the item) occur.

Transformation denotes an Event in which an entity neither gains nor loses attributes, but one or more attributes are altered. This could represent a character dying (state of being changing to dead) or getting older (age being replaced by an incremented value). This is in essence a combination of Gain and Loss, as the entity loses an attribute and gains another, but it is useful to provide a condensed version as Transformation is used frequently, especially to signify movement. Finally, Introduction is a comparatively straightforward Event which marks the first appearance of an entity in the medium.

Each event may have preconditions and postconditions specified, which respectively denote the conditions required for the event to take place or be judged as complete. For example, if a character is to gain an item, the giver must be in possession of the object for the event to occur. These properties are useful both for the analysis process (e.g. locating flaws in continuity) and the annotation process (e.g. if certain conditions are met, some events may have more chance of occurring).

5.4 Extensibility

5.4.1 Names

OntoMedia provides a set of classes specifically for the description of names. At the lower level are the seven Name-Component subclasses, which subclass Name-Component and provide a reference to textual data. These subclasses include Title, Family-Name, Primary-Name (e.g. the first name), as well as others for foreign names and qualifiers. At the next highest level is the Name-Ordering class, which simply specifies the order in which the Name-Component objects should appear. This has subclasses for Formal, Informal, Familiar, and Full names, although these are primarily for querying purposes. For example, Russian names have a different form to English names, and yet can still be classed as Formal or Full.

5.4.2 Geometry

To ease the markup of portions of images, or areas of locations, a set of Geometry classes are available. These make use of a set of Point primitives, which are subclassed to allow for 3D Points, pixels, and GPS co-ordinates. There is a further Distance class to specify a length with an associated Unit, which may be extended further to allow reuse. Finally, there are a collection of general shape classes, including Circle, Ellipse, Rectangle, Line, PolyLine, and Polygon.

5.5 Case Studies

Contributing to the cross-modal capabilities of OntoMedia was a set of three different motivations. In order to support the needs of the developers these motivations were kept in consideration, and the result is an ontology which provides a highly portable core infrastructure and the specializing ontologies described earlier.

5.5.1 Applying to Fiction

At a base level, fiction shares many of the same characteristics as film and other narrative media. Specifically, a set of entities (characters and items) interact during events in the story. Fictional narrative is not as dependent on time as film and factual events, but a sense of chronology may still be maintained. The primary goal for the application of OntoMedia to this area was the creation of a recommender system for online fiction with the ability to provide summaries of the story while avoiding elements which readers felt could spoil the storyline. For example, this could include the revelation of character deaths (especially if major characters).

To extend OntoMedia to handle this case required two extensions to the ontology - namely classes to flag the 'spoiler' elements of a story, and classes to indicate the extra content within online fiction. Furthermore, the 'Context' class was created to allow for the separation of fiction into different universes, as many authors might write in a science fiction or fantasy genre using characters from preexisting material. The Context representation provides the ability to reference other characters present in the universe of the fiction.

5.5.2 Applying to Film

This case was of primary importance to the SBS project, as it allows for the representation of elements within a movie. While it is possible to annotate fiction with OntoMedia (as described earlier), film requires a few extra elements. Firstly, it combines classes

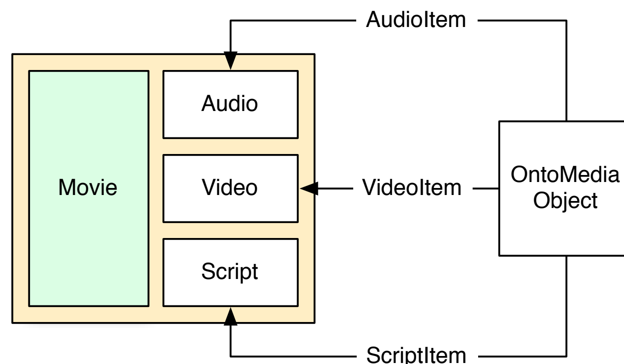


Figure 5.7: Binding OntoMedia objects to other media

from both narrative aspects and fictional aspects, with characters portrayed by beings (the actors). This allows for ties to information on the actor, such as pertinent features which may be used for automatic annotation. In some cases the character is portraying another character or being, and this is also supported in the ontology. Film is closely tied to time, and the location specifier classes built for OntoMedia allow for Occurrences to be bound to a set span of film - with participating entities defined as polygons within the frames.

The audio track of film is also supported within the OntoMedia definition. This is where dialogue and foley effects are present, so these may be tied directly to events or character entities. AudioItem, which subclasses the MediaItem class, provides for this, with subregions allowing for the subdivision of audio where a single audio track file is provided. These regions, themselves AudioItems, may then be specified to relate to other items, such as the corresponding video portion.

Through the use of the previously described SiX format, it is also possible to bind events and characters to the corresponding script portions. A ScriptItem element provides a reference to the SiX resource, with subregions able to reference sections using the ID defined in the annotated script. Figure 5.7 illustrates the available interlinks between media using OntoMedia.

5.6 Summary

The OntoMedia project provides a powerful set of classes for media annotation, with its extensions for script and film particularly relevant to the State-Based Sequencer. It is also highly extensible, while always providing the core temporal and spatial classes for high-level comparison. Its application to fiction and narrative annotation projects is a

significant example of its ability to handle a variety of media formats, with this portability and extensibility providing the possibility of dialogue and camerawork annotation as future sources to the system.

Chapter 6

Composer Representation

While the OntoMedia representation provides a formalised version of a directing medium, the SBS composer representation provides the essential mapping between the concepts in the source media and the resultant music. The mapping is defined by an RDQL (RDF Data Query Language) query which is responsible for determining the positions and durations of 'musical modifiers'. These are collections of parameters which modify the inputs (e.g. pitch graphs) used by the composition agents, with each agent responsible for processing its specific inputs. This chapter describes the modifiers defined by the composing agents, as well as how these parameters are bound to the medium using RDQL queries.

A composer is defined as a set of pairs: the first element specifying a situation within the directing medium, and the second specifying what modifications should be made to the soundtrack. As an example, it may be the case that a certain composer wishes to use saxophones whenever an event occurs within a room with blue walls. The situation would be defined in RDQL and would produce all events in which a room with blue walls occurs. The modification would specify that the probability of a saxophone being chosen as instrumentation should be increased. The other parameters available for modification are described later in this chapter.

The composer representation is an entirely novel approach to binding annotated media and musical parameters which was designed by the author specifically for the SBS project.

6.1 Musical Modifiers

All modifiers in SBS are placed according to 2 parameters: a start point and an end point defined in frames. If no end point is specified, the modifier is placed as a single-point event, which is useful for cases such as tempo where beats could be inserted at

Tempo	BPM Range
Largo	40-60
Larghetto	60-66
Adagio	66-76
Andante	76-108
Moderato	108-120
Allegro	120-168
Presto	168-200
Prestissimo	200-208

Table 6.1: Approximate BPM ranges for standard tempos.

specific locations. The frame values are converted into seconds, and then beats, by the tempo agent (described in more detail in Chapter 8).

At the root level of a modifiers structure is a <modifiers> tag with an accompanying 'name' attribute. This acts both as a grouping for several modifiers and as a means to reference the set of modifiers from the mapping section. As such, the name should be unique. Some modifiers may also use the 'affects' attribute, which provides a comma-separated list of instruments affected by the modifier. The <modifier> tags inside this element may have two attributes: type and mode. Type specifies the type of agent at which the modifier is targetted, and mode specifies the context of the modifier. For example, tempo is available in 'bpm' and 'beat' modes, and key is available in 'scale' and 'change' modes.

From here, the content of the modifier may vary from agent to agent. However, some tags are used consistently to ease parsing. A <sequence> tag provides an ordered list of items, such as pitches or beats, and a <note> tag with 'pitch' and 'length' attributes describes a single note.

6.1.1 Tempo

Tempo, which indicates the speed of a piece of music, is the attribute which is most dependent on the high-level view of events occurring in the medium. For example, a scene with dancing should be set to a tempo which coincides with the action.

SBS defines all tempos as being in BPM (beats per minute), with some 'standard' tempos having preset values. For example, 'presto' (fast) is defined as being between 168 and 200, while 'prestissimo' (very fast) is defined as being between 200 and 208 (see Table 6.1). To allow for this, a BPM parameter is available and can be weighted appropriately (see Listing 6.1).

If it is preferable to choose a tempo depending on the content of a scene, a beat may be inserted at an appropriate point (see Listing 6.2). For example, it could be specified that

```
<modifiers name="allegro">
  <modifier type="tempo" mode="bpm">
    <value>120</value>
  </modifier>
</modifiers>
```

Listing 6.1: An example tempo modifier

```
<modifiers name="gunshot">
  <modifier type="tempo" mode="beat">
    <value>1</value>
  </modifier>
</modifiers>
```

Listing 6.2: An example tempo modifier using a beat placement

```
<modifiers name="waltz">
  <modifier type="pulse">
    <sequence>
      <beat strength="1" />
      <beat strength="0" />
      <beat strength="0.5" />
    </sequence>
  </modifier>
</modifiers>
```

Listing 6.3: An example pulse modifier

a beat event should be present on all gunshots or explosions by inserting beat markers at the onsets of the appropriate OntoMedia Action element. Alternatively, the mapping may indicate that beats should *not* occur at certain locations. The tempo agent then calculates a tempo with the BPM range, or creates a new tempo if no BPM is provided.

6.1.2 Pulse

To represent pulse, a sequence of beats is created. These beats have strength values ranging from from 0 to 1, with 1 being the strongest and 0 the weakest. In a musical context, 0 is a 'regular' beat, 1 the first beat (the downbeat) of the sequence, and 0.5 could represent the final beat (or the upbeat). A waltz pulse could therefore be described as '1, 0, 0.5'. The pulse starts on a beat determined by the tempo, repeating until a change is triggered.

6.1.3 Rhythm

As with pulse, rhythm is represented as a sequence, but in this instance a sequence of note lengths is used. Note lengths are numeric, using 4 as a semibreve (a whole note), 2 as a minim (a half note), 1 as a crotchet (a quarter note), 0.5 as a quaver, and 0.25 as a semiquaver. In this instance, however, a note can be specified as a rest by setting the 'rest' attribute to 1, in which case no notes are played during that time. As numbers are

```
<modifiers name="dotted_minim">
  <modifier type="rhythm">
    <sequence>
      <note length="3" />
      <note length="1" />
    </sequence>
  </modifier>
</modifiers>
```

LISTING 6.4: An example rhythm modifier

```
<modifiers name="major">
  <modifier type="key" mode="scale">
    <sequence>
      <pitch pos="1" />
      <pitch pos="3" />
      <pitch pos="5" />
      <pitch pos="6" />
      <pitch pos="8" />
      <pitch pos="10" />
      <pitch pos="12" />
    </sequence>
  </modifier>
</modifiers>
```

LISTING 6.5: A major scale definition

C	C#	D	D#	E	F	F#	G	G#	A	A#	B

FIGURE 6.1: C major scale shown using notes of a chromatic scale.

used to represent note lengths, it is also possible to use dotted lengths, with a dotted minim being 3 (a minim plus a crotchet), and triplets, where (for example) three notes could span the duration of 2 crotchets by each being of duration 0.667. To allow for variation, it is recommended that single note lengths also have a slight weighting, as this gives the possibility of breaking out of preset rhythms for emphasis. Listing 6.4 shows the modifier for a dotted minim / crotchet rhythm.

6.1.4 Scale and Key

Scale and key are closely linked, with a scale describing the intervals between pitches and the key adding a starting note to establish the pitches available. The scale definition is specified using the semitones of a well-tempered chromatic scale (although it could be extended to support microtones by using fractional position values). The first semitone in the scale is 1, with the final being 12. Listing 6.5 shows the representation of a major scale, with Figure 6.1 illustrating how this would describe a C Major scale.

While the scale above specifies the basic shape of the scale, the key also includes the starting note. The representation used by SBS is very simple in this instance, with only three parameters: the unique name, the root note (stated as a note name), and the scale

```
<modifiers name="C">
  <modifier type="key" mode="key">
    <root note="c" />
    <scale ref="major" />
  </modifier>
</modifiers>
```

Listing 6.6: A definition of C Major

```
<modifiers name="relative">
  <modifier type="key" mode="change">
    <change start="C" end="Am" reversible="true" />
  </modifier>
</modifiers>
```

Listing 6.7: Defining C Major's relative minor

(using the unique name defined in the scale section). C major, for instance, would use 'C' as the root note, 'major' as the scale, and 'C' as the key name (see Listing 6.6).

The progressions between keys, or key changes, are also defined in the composer representation. This specifies the starting key, ending key, and whether the change is reversible (such as major to minor and minor to major). This can be seen in Listing 6.7 (The 'Am' key would be defined in the same way as with 'C').

6.1.5 Chord

The chords and chord movements in a piece are vital if the harmony is to appear natural. As well as providing underlying motion, chords provide both transition into new keys and terminating clauses (known as 'cadences'). In homophonic (or 'chordal') music the notes of a chord are typically sounded at the same time, as is evident in Bach chorales, whereas in polyphonic music the notes may be separated. As such, chords also make up the texture of the music between the bassline and melody. Two sections are provided in the composer representation for chord description, with one specifying the constituent notes and the other the transitions between chords.

To provide a scale-independent representation of chords, no pitch names are used in their definition. Instead, note numbers are specified, with note 1 corresponding to the first note of the scale, and so on up to the end of the scale. This notation means that it is possible to represent a simple triad as '1', '3', '5' - a chord which can be major or minor depending on the scale that is chosen. This is also transposition independent, with the previous chord being valid starting on any pitch.

Order is essential in the chord definition, as the first note indicates the bass note, the second the next note above that, and so on. This allows for the idea of 'inversions' to be preserved, with the previous chord being the first inversion of a triad, and '3', '5', '1' being the second inversion. As with scales, each chord has a unique name. These are

```
<modifiers name="I">
  <modifier type="chord" mode="notes">
    <sequence>
      <note pitch="1" />
      <note pitch="3" />
      <note pitch="5" />
    </sequence>
  </modifier>
</modifiers>

<modifiers name="V">
  <modifier type="chord" mode="notes">
    <sequence>
      <note pitch="5" />
      <note pitch="7" />
      <note pitch="2" />
    </sequence>
  </modifier>
</modifiers>

<modifiers name="perfect_cadence">
  <modifier type="chord" mode="progression">
    <sequence>
      <chord ref="V" />
      <chord ref="I" type="cadence"/>
    </sequence>
  </modifier>
</modifiers>
```

Listing 6.8: An example chord modifier

ideally suited for standard chord notation, such as Ic for a chord starting on the first note of the scale in the 3rd inversion.

The second set of definitions relating to chords are progressions. As mentioned, these specify the transitions from chord to chord, and can be cadential (i.e. ending a phrase). Progressions consist of a list of chords, referenced by their unique names, in the order that they should appear. The ‘type’ attribute can be set to ‘cadence’ to indicate a cadential progression.

6.1.6 Instrumentation

The instrumentation of a piece is more complex than just the kind of instrument being used, as both the range and dynamic must be taken into account. All instruments have a range of pitches which are musical, but some composers choose to go outside these ranges to affect the music. Furthermore, instruments can be overblown, or played harshly, which is ideal to instill a sense of shock or worry. The SBS representation allows for the definition of instruments, together with valid ranges of pitch and dynamic. These can then be weighted by events within the script.

The most basic instrument representation is the instrument tag itself. This contains the unique instrument name which is, for consistency, comprised of the instrument category followed by the name of the instrument. So, a trumpet could be specified as ‘brass:trumpet’ and a xylophone could be specified as ‘percussion:tuned:xylophone’.

```
<modifiers name="string:violin:bowed">
  <modifier type="instrumentation">
    <dynamics name="dynamics:pp" from="73" to="84" />
    <dynamics name="dynamics:p" from="85" to="94" />
    <dynamics name="dynamics:mp" from="95" to="102" />
    <dynamics name="dynamics:mf" from="103" to="109" />
    <dynamics name="dynamics:f" from="110" to="115" />
    <dynamics name="dynamics:ff" from="116" to="121" />
    <pitch name="pitch:g" from="55" to="61" />
    <pitch name="pitch:d" from="62" to="68" />
    <pitch name="pitch:a" from="69" to="75" />
    <pitch name="pitch:e" from="76" to="87" />
  </modifier>
</modifiers>
```

Listing 6.9: The instrument definition for a bowed violin

This also allows for expansion after the name, such as in 'brass:trumpet:muted'. A future possibility could be the creation of an instrumentation ontology, which would allow for a standard representation of the instruments' attributes.

To focus the instrument definition further, 'dynamics' and 'pitch' tags are available. These are specified as ranges with unique names, with the minimum and maximum values using MIDI values (i.e. from 0-127). Listing 6.9 shows the instrument definition for a bowed violin, and includes example dynamic settings.

6.1.7 Melody

While the scale and key information provide the 'base' information for the notes of a piece, the melody modifiers allow for weighting to be placed on certain combinations of notes. For example, intervals of fifths in a melody often invoke a sense of heroism (such as in Star Wars) and chromatic intervals (i.e. moving by semitone) typically suggest unease, with Jaws being the oft-quoted case. Of course, the composer may wish to specify longer phrases than a single interval, so the melody modifer is described (as with the chord modifier) as a sequence.

Each note within the melody modifier is specified as a number, which refers to the position within a chromatic scale. For example, an interval of a fifth would move from note 1 (the root note) to note 8 (the fifth), while a chromatic sequence could move from note 1 to note 2 (the semitone above). As with the scale definition, fractional values could allow for microtonal information in a future extension. This representation of pitch allows for a key-independent description, which will be subsequently weighted by the scale (so a minor scale plus a chromatic melody may both be applied to the same event).

```
<modifiers name="major_arpeggio">
  <modifier type="melody">
    <sequence>
      <note number="1" />
      <note number="5" />
      <note number="8" />
    </sequence>
  </modifier>
</modifiers>
```

Listing 6.10: An example melody modifier

```
<mapping>
        <map>
        <query>
        (?eve ome:has-occurrence ?occ)
        (?eve ome:has-location ?loc)
        (?loc rdfs:type space:Corridor)
        </query>
        <promote>
                <param ref="strings" />
        </promote>
        </map>
</mapping>
```

Listing 6.11: Promoting the use of strings when an event takes place in a corridor.

6.2 Binding to a Semantic Annotation

Once the medium is marked up and imported into a triple store, it is necessary to map from this into the composer representation. To ease this process, a simple XML format contains an RDQL 'WHERE' fragment expressing the item in OntoMedia format as well as the properties to apply to the resultant music. Listing 6.11 shows an example case, in which strings are given priority in events set in corridors.

The translator uses a three-stage approach to the construction of the landmark file (see Figure 6.2). First, every available event is retrieved from the triple store, together with frame information and ID. This is used to create slots for each event into which modifiers may be placed. Even empty event slots may be useful for tempo estimation, so these are kept in the resultant landmark file. Next, every query in the mapping is applied to the triple store, and the result is a list of event identifiers which correspond to those of the event slots. For each query which matches, the appropriate modifiers are added to the event slots.

Finally, the promote and demote modifiers are combined where possible. Every promote operator increments the probability of the modifier being applied, and every demote operator decrements the probability (with 0 as a minimum). So, if two mappings indicate that strings should be used for the event's music and one suggests that brass be used, the strings will have a 2/3 probability of being chosen and the brass a 1/3 probability. Listing 6.12 shows an example portion from a landmark file. Note the modified form

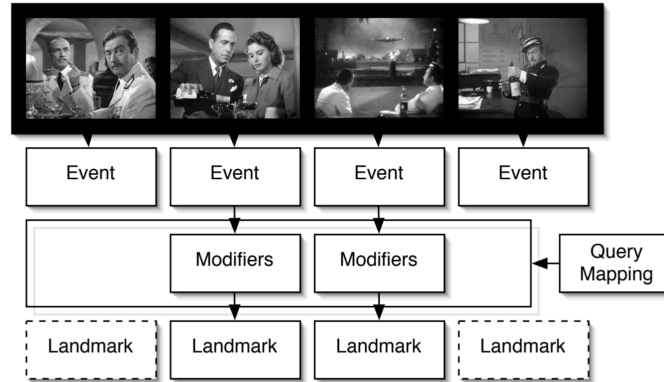


FIGURE 6.2: Mapping from a composer representation to a landmark file

```
&lt;segment from="1000" to="5000"&gt;
  <landmark at="0">
    <modifier type="melody" ref="major_arpeggio" probability="0.25" />
    <modifier type="melody" ref="minor_arpeggio" probability="0.75" />
    <modifier type="pulse" ref="waltz" probability="0.6" />
    <modifier type="pulse" ref="march" probability="0.4" />
    <modifier type="rhythm" ref="dotted_minim" probability="0.5" />
    <modifier type="rhythm" ref="dotted_crotchet" probability="0.5" />
    <modifier type="instrumentation" ref="strings" probability="0.33" />
    <modifier type="instrumentation" ref="woodwind" probability="0.67" />
  </landmark>
&lt;/segment&gt;
```

LISTING 6.12: A portion of a generated landmark file.

of the modifier tag, with an additional probability attribute and a type to ease unique naming.

6.3 Summary

The SBS composer representation has two purposes: the description of a composer's technique when writing music for specific concepts, and the binding of these descriptions to the OntoMedia annotation using RDQL mappings. Musical modifiers are designed such that parameters may be customised for specific agents while the XML structures are reused, hence reducing the need for extra parsing functionality. Furthermore, the promote tags allow representations to suggest modifiers without any probability values, as these are added at the translation stage. The resultant landmark file is passed as an input to the Light Agent Framework and the SBS composing agents which are detailed in the next two chapters.

Chapter 7

The Agent Framework

7.1 Introduction

In order to connect and test the various algorithms within the composing system it was necessary to implement a framework in which the components could be coupled. Furthermore, it was decided that the framework should be portable, providing a simple protocol (see Table 7.1) for agent registration and communication, and lightweight, hence requiring a minimum of configuration and resources. The Lightweight Agent Framework was built with these goals in mind, and it has been developed to a state where it has been used by both SBS[1] and by the entirely different application area of multi-camera image sequence construction[2,3]. Both of these applications are cluster-based, so the use of a succinct network protocol was ideal, and neither rely on intelligence in the agent communications, so the basic agent was employed for the task. The Java implementation of LAF, the Launchpad agent, the agent graph implementation, and the plugin-enabled router are entirely my own work, while the design of the framework and its associated protocols were a collaboration between myself and Lee Middleton. This chapter gives an overview of the Light Agent Framework's implementation and how it may be used to create networks of dedicated agents.

7.2 Agent Design

Every agent used within the Light Agent Framework has a central engine, responsible for carrying out its task. An engine may only be activated by one client at any one time to preserve atomicity, and hence two messages (LOCK and UNLOCK) are present

[1] Jewell et al. [2005b]
[2] Middleton et al. [2005b]
[3] Middleton et al. [2005a]

Message	Description
IDENTIFY	Sends an agent stub to the router.
IDENTITY	Requests an agent stub from a router.
LOCK	Locks the next available agent of the specified type.
UNLOCK	Unlocks a locked agent.
SETPORT	Sets an agent's port value.
GETPORT	Retrieves an agent's port value.
CALL	Executes the agent's engine.
FLUSH	Clears all of an agent's ports.
SUBSCRIBE	Connects an agent to the router.
UNSUBSCRIBE	Disconnects an agent from the router.
MONITOR	Sets up a listener on one or many ports of one or many agents.
NOTIFY	Sent by the router to indicate a change in an agent's ports.
LOGGER	Sets up a listener for log messages of a certain level from an agent.
LOG	Sent by an agent to provide informative log messages.
KILL	Disconnect the agent to which the message is sent.
PING	Test the connection status of an agent.
SHOW	Requests a list of all available agents.

Table 7.1: The message types available to the Light Agent Framework. All return OK or NOK on success or failure.

to allow for this. Using separate messages for this process allows for the handling of multiple parameters, as these may not all be available at the time of calling.

All agents within the framework connect to a router (several may be available) via autoconf, removing the need for network-related configuration. Once connected, the agent sends a 'stub' detailing its type, as well as other agent-specific information. This is used by the router to select agents: a client requests an agent of a certain type, and the router returns the unique ID of the first unlocked agent of that type. If a client chooses to use the agent, it locks it, sets any necessary parameters, triggers the engine, and finally extracts the output and unlocks it. During this time no other client may make use of the agent unless the client disconnects unexpectedly. Most importantly, only the locking client can alter the input and output parameters of a locked agent. To all other clients, the agent seems immutable. The following sections detail this process in more detail.

To aid in the abstraction of communication between a client and an agent, LAF supports a Remote Agent concept. When an agent is locked, the framework passes back a remote agent object which has several convenience methods, including setInputPort, getOutputPort, and call. These handle the setting/retrieval of parameters and the execution of the engine process.

```
<identity>
  <type>string.concat</type>
  <creator>Mike Jewell</creator>
  <description>Concatenates two strings together</description>
  <version>1</version>
  <ports>
    <port direction="input" name="a" optional="false" type="string"/>
    <port direction="input" name="b" optional="false" type="string"/>
    <port direction="output" name="c" optional="false" type="string"/>
  </ports>
</identity>
```

Listing 7.1: The external stub file for a string concatenation agent

7.2.1 Identification

The agent stub is similar in purpose to the CORBA Interface Definition Language[4], as it provides an interface to the agent that both the client and server can use easily, regardless of platform. Where it differs from IDL is in its simplicity, with only five sub-elements present. The first four of these are used with the agent identity ports, and consist of the type, creator, description, and version. The final element describes the ports of the agent, with 'type', 'direction', 'name', and 'optional' attributes per port. Agents can be initialised using an external stub file, thereby easing the code required in the agent constructor. Listing 7.1 shows an example stub for a string concatenation agent which takes in two strings and outputs the first string joined to the second.

When agents connect to a router and subscribe, they can optionally send an IDENTIFY message. This provides the stub to the router, which it can then supply to other agents when an IDENTITY request is submitted. This also reduces the amount of data transmitted by the SHOW message, which simply returns a list of agent names rather than a list of stubs. A client can then use IDENTIFY messages to request more detail.

7.2.2 Ports

The parameters of agents in the Light Agent Framework are represented using 'ports', which are effectively slots which may be connected from one agent's output to another agent's input. Ports have specified types, and are denoted by a unique name to ease referencing. Each port is placed in a loose hierarchy, with '.'s separating the parent of a segment from the child. Five port types are provided by default, namely agent.identity, agent.input, agent.output, agent.state, and agent.call.

The agent.identity port is responsible for containing the information sent to the router in the initial connection phase. As detailed eariler, this includes the type, creator, and description fields. As with ports, the type of an agent may be delimited with '.'s, such as 'string.concat' or 'music.composing.genetic'. The 'agent.input' and 'agent.output' ports

[4] Framingham [1999]

are respectively responsible for the input and output values of the agent, with the parameter name appended to the relevant port prefix. For example, agent.input.landmarks and agent.output.musicxml are valid cases.

agent.state and agent.call provide more low-level information on an individual agent. agent.state gives the current agent status, and is 'waiting' upon initial creation, 'ready' when all necessary ports are set, 'running' when executing, and 'exiting' upon completion of the execution process. agent.call is reserved for use by the agent itself, with several ports for progress information: agent.call.percentage provides a completion percentage, agent.call.time.current gives timing information for the current task, agent.call.time.total accumulates this for the entire process, and agent.call.status is a string port for any other useful status information.

7.2.3 Monitors

Rather than polling the content of individual agent ports, LAF supports a monitoring technique. A client can request to monitor an individual or a subset of the ports available, and a callback is triggered automatically if that port changes. This is made possible with two messages: MONITOR and NOTIFY. The former requests that the router monitors a set of ports on an agent, while the latter indicates that a change has occurred.

Wildcards can be specified in both the agent and port name when specifying a monitor, with agent.input.* indicating that all input ports should be observed. To monitor the output ports on all string concatenation agents, 'string.concat.*:agent.output.*' would be used.

The monitoring capability is at the heart of the agent graph structure, described later, as it is possible to check the inputs and outputs of the nodes within the graph.

7.3 Router Design

As mentioned previously, the LAF router is modular in design. Several plugins were implemented for the Java router, including a logging plugin, a monitor plugin, an identification plugin, and a state plugin. The first three of these correspond to the LOGGER/LOG, MONITOR/NOTIFY, and IDENTITY/IDENTIFY messages. Respectively, these message pairs allow for the transmission of logging information, notification information on a port change, and agent stub details. The abstraction of these messages into removable components allows for a very lightweight router for circumstances where resources are limited. Finally, the state plugin is responsible for keeping an accurate representation of the state of the router, such as which agents are connected

and the states of these agents. This is primarily for debugging and audit trails, but can also be useful for web-based status monitoring.

Further to these plugins, the router implementation uses a 'selector' module. This specifies which agent should be selected when a client requests a type. The base model in LAF is that of the locking selector. This handles the LOCK/UNLOCK messages, and locks the next available unlocked agent in order of their subscription. This could be extended to allow for resource or platform checks. The latter case is especially suited for the launching of agents on machines with sufficient resources.

In summary, the basic router only handles subscription messages, disconnect messages, and routing itself. It is through the use of plugins and selectors that features can be added and as such the router can be tailored to suit the application.

7.4 Agent Graphs

While the addition of individual agents to an agent network is essential to the functionality of the system, this behaviour is not ideal when you wish to join several agents together. To ease this, LAF includes functionality for building 'agent graphs'. These are structured as directed graphs, with edges connected between the ports of individual agents. The strong typing of the agents is essential here, as it ensures that a connected agent will receive parameters which are suitable for its engine. On execution, the agents in the graph with all parameters filled are locked and executed and, on completion, the input ports of the connected agents are set. This repeats, with each succesive agent executing, until no agents are left. Figure 7.1 shows the flow of information from agent to agent in an agent graph.

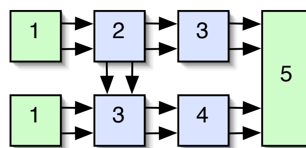


FIGURE 7.1: An example agent graph, with numbers indicating the order of execution given identical agent execution times.

One difficulty to be overcome during the creation of the agent graph implementation was that of the discovery of agent input and output requirements. If the addition of an agent to a graph required the existence of an instantiated agent in the system, construction of the graphs would not be possible in an offline environment. However, the agent stub system described earlier is ideal for this situation, as the stubs may be stored on a local machine and they provide information on input and output ports, as well as other metadata. So, all agents designed for SBS are bundled with an accompanying stub XML file to allow for offline creation of the agent graphs.

7.5 Agent Launcher

On some occasions it will not be possible to execute an agent graph, as the agents will not be available for use. To remedy this an agent launcher was developed. This is itself an agent, with a single 'class' port, to which the binary of the agent class is sent. Once executed, the launcher launches the class on the machine on which it is running. This structure allows for the option of a network of launcher agents, with the required agents executed by the launchers selected by the router, possibly depending on resource availability.

7.6 Summary

The Light Agent Framework was developed from the ground up as a portable, standardized, system, and as such is ideal for the handling of musical agents. Versions have been developed in Java, C++, and Python, all following the same API to ensure compatibility, so there is a rich selection of programming environments available. Furthermore, the design of the framework allows for a strong level of extension, with much of the system being modular. The following chapter provides details of the music agents, along with how the Light Agent Framework's features were integral to the functionality of the project.

Chapter 8

Agent Designs

8.1 Overview

8.1.1 From Algorithm to Agents

Fundamental to the operation of the State-Based Sequencer was the ability to encapsulate the functionality of an algorithm into one or many agents. These agents would use a common interface to allow for straightforward replacement and testing, while ensuring the algorithm operation was not impaired as a result. For this purpose, a standard musical agent was developed (see Figure 8.1). This takes two inputs: one to describe parameters and one as a reference to the current piece of music. The agent parses these into the relevant data structures, and then carries out the code in the 'agent engine'. This could be the evolutionary stages of a genetic algorithm, the iterative process of grammatical composition, or any other technique suited to the task of the agent. After the engine has completed, the results are encoded back into modified parameters and music and passed out of the agent.

The parameters of the composing agents are passed in the form of the landmark file, described in Chapter 6, and are provided as an XML string. The agents are free to handle this in a manner suited to their task, as it may be possible to process segments irrespective of their temporal placing (i.e. giving the possibility of parallel handling) or the timing may be essential (e.g. in the case of a motif, where it can develop over the whole score). Agents may also add or modify the parameters and pass through the altered file for the next agents in the graph. This is required in cases where a probability mesh may be refined to a single option, such as converting a set of pulse probabilities into the exact pulse placements for use by the rhythm agent. The design of all of the SBS agents, as well as the parameter XML file passed to the various agents, is entirely the work of the author.

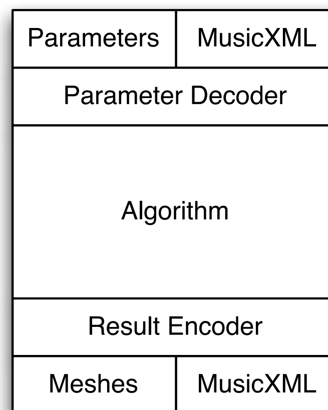


Figure 8.1: The standard agent template employed by SBS

The musical information is passed in the form of a MusicXML file. This format was defined by Good [2001] and provides a high level of detail when representing music. MusicXML was not designed primarily for performance, instead intending to give enough information for the rendering of a notational version of the piece. In comparison, MIDI *was* designed primarily for performance, and hence defines only the instructions necessary to handle note playback and instrument selection. SBS does not focus on the performance of the final piece, prefering to delegate the task to performers or a performance system, and as such MusicXML was an ideal choice. Other formats for notation representation were also considered, but were all deemed to have been largely superseded by MusicXML due to its open, easily-parsable, format.

8.1.2 Designing Agents for Musical Composition

Every agent in the SBS system has a similar implementation. At the base level is the agent itself, which subclasses the Light Agent Framework's Agent class. This connects to the router, provides the relevant identification stub, and waits on activation. Once called, the landmark and MusicXML parameters are parsed into suitable objects for the agent's engine which is then executed. On completion, the landmarks and MusicXML are retrieved from the engine and passed out via the respective ports.

The engine itself extends the base MusicEngine class. This provides a set of default methods for the setting and retrieval of the landmark and MusicXML parameters, as these are common to all of the engines. The MusicEngine also provides a 'run' method which is overridden by the extending classes. This is where the agent-specific functionality takes place, and is the subject of this chapter. Three utility classes are present, with the LandmarkFile class including functionality to read and write landmark files,

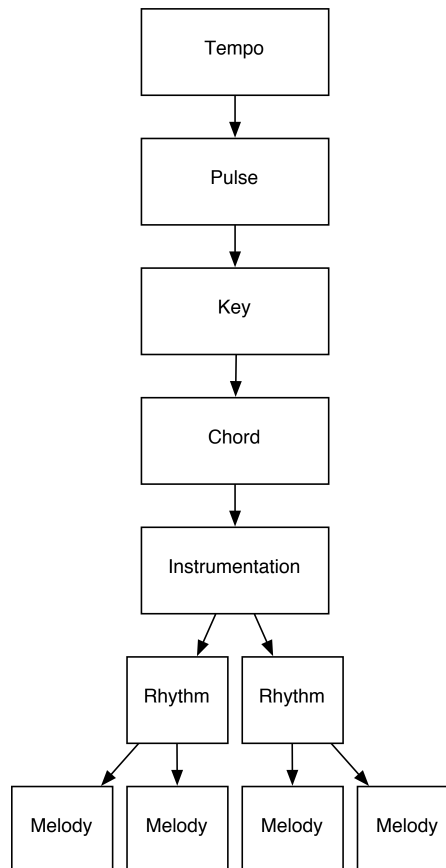


FIGURE 8.2: The SBS Agent Arrangement

the Landmark class encapsulating the parameters contained within this file, and the MusicXML class handling the parsing and editing of MusicXML files.

8.1.2.1 Specialising for Genetic Algorithms

The Genetic Algorithm Agents (GAA) in SBS make use of the Java Genetic Algorithms Package[1] (JGAP) developed by Neil Rotstan. Using an object oriented approach to separate the various entities involved in GAs, this package provides representations of genotype, genes, chromosomes and fitness functions, as well as several operators (including crossover and mutation) and several selection techniques (including tournament and roulet selection). As such, it is a useful and well-documented foundation for the development of the GAAs.

The GAAs have two classes on top of the standard agent classes which extend JGAP's FitnessFunction and Gene. The FitnessFunction class requires a single 'evaluate' method, which returns a floating point value representing the fitness of a single chromosome. The evaulation function varies depending on the agent, but all take into consideration the

[1] http://jgap.sourceforge.net/

landmarks passed to the agent and the individual genes of the chromosome to calculate a suitable score.

The Gene class represents the smallest element of a chromosome, which again is customised depending on the agent. For example, the tempo agent uses Boolean values to represent beats, while the pulse agent represents beat strength as floating point values. The Gene classes include comparison functions, mutation functions, and methods to obtain and set the gene's allele.

8.1.2.2 Philosophy

When deciding how individual agents should operate, the first reference was to a composer's technique. A film composer would start with a blank score, with cue marks then added from an initial screening of a film as well as some notes as to possible musical features. The tempo agent provides the cue marks, and the landmark file the musical hints. From here the composer may decide on the pulse of the music - whether it is a waltz, whether it should have syncopation, etc. This stylistic choice is pertinent to all of the instruments, hence its placement early in the agent chain.

The key is the next requirement, as this is again common to all instruments and is fundamental to the mood of a scene. Musical theory, with support from psychological experiment[2], suggests that certain progressions between keys are perceived as more 'natural', such as relative major/minors or those linked via the circle of fifths. Using a graph structure for this is therefore ideal, as it allows for these relationships to be weighted - or not weighted if the composer wishes to experiment with other transitions. The Genetic Algorithm for this sticks strongly to this graph, so it is unlikely that unexpected key changes will occur.

From the key, the chord progressions naturally follow. Again, these are common to all instruments (although some may choose to diverge from the norm), and again some chord progressions are naturally comfortable to hear. For example, cadences such as the perfect cadence (from the fifth chord in a key to the first) and the plagal cadence (from the fourth chord to the first) provide a sense of completion to a passage, and are often used at the end. Inversely, the interrupted and imperfect cadences suggest that the piece is not yet finished. Finally, there are some sequences of chords that are commonly used, such as moving from the tonic (I) to any other chord, or I IV V, often used in the 'three-chord song' which underpins much simple blues and rock and roll. Therefore, as with the key agent, a graph approach to transitions is ideal, with breaks from this graph rare (unless requested).

Once the composer has the harmonic ideas in mind, the instruments are selected. These are again significant to how a scene is perceived, but the composer will also aim to have

[2] Maess et al. [2001]

an ensemble that supports the film. For example, a string quartet playing smoothly would work well with woodwind, but it may be beneficial to use brass if the strings were playing accented notes. At present SBS uses a simple approach, simply selecting the most weighted instruments, but it would be possible to use the pitches provided by the landmark file to choose. Furthermore, the landmark file could be extended to support 'effects' such as staccato and accents.

The branch into instrument-centric agents begins with the rhythm agent. In this case it is less useful to use a graph with nodes linked, as composers are more likely to use repeated rhythmic patterns. As such, the genetic algorithm instead scores using a weighted set of segments. This has a much higher chance of producing unexpected results than the key/chord agents, and adapting the GA fitness threshold allows a range from entirely random rhythm to those which reproduce the given segments with no deviation. Future algorithms could use a specialised graph model, with the nodes consisting of the segments and with transition rhythms between different options. Another alteration could 'trigger' transitions via the landmark representation, hence only changing rhythm at certain positions.

Finally, the melody of the piece is handled. This would typically go together with the rhythm when composing traditionally, but the process is so different from the rhythmic process that it is separated in SBS. Here, the composer aims to write a melodic line, possibly repeated, which conforms loosely to the key signature while being allowed to deviate from these notes on occasion. Furthermore, the composer tries to cover the main notes of the chord, although again this can vary depending on the composer's intentions. As such, a graph-based fitness function is again useful - the structure allows for the representation of the transitions between notes, while weightings can ensure that the intervals follow the composer's guidelines. In addition, weightings on the nodes themselves allow for pitches to be weighted - which is essential for instruments with limited pitch ranges.

8.2 Tempo Agent

The tempo agent makes use of the segmented structure of the landmark file to specify where tempo changes should be. Each segment can be handled individually, so easier cases can be optimized for speed. First, segments that have exact tempos with no extra beats are processed. As it is not necessary to calculate the BPM, they can be immediately converted into separate markers (one per beat). Next, the agent handles segments that have no single tempo specified, but have beats in place. In this case, a set of non-unary factors is calculated with members coinciding at the beat locations on as many occasions as possible. For example, if beats occured at frame 30, 40, and 55, this set would contain 5 as the best factor, followed by 10. If any of these values are

present in the histogram, they are preferred. This is then converted into a BPM value, and markers are set per beat as before.

Finally, segments with no tempo indications are handled. These are much less likely to appear, as the mapping will usually specify a 'default' tempo. This case is handled with a straightforward approach:

1. If there is a preceding tempo indication, this is carried forward.
2. If there is no preceding tempo, but there is a succeeding tempo, this is brought backward.
3. If there is neither a preceding nor succeeding tempo, the tempo defaults to 120bpm.

8.3 Pulse Agent

Chromosome Encoding: String of floating point numbers reprsenting beat strength.

Genetic Operators: Crossover (1/5 probability), mutation (1/8 probability), sequence (1/8 probability).

Fitness Function: Using provided beat option sequences, scored on closeness of match multiplied by weighting of sequence.

Population Size: 1000

Selection Approach: Threshold selection

The pulse agent makes use of the beat markers from the tempo agent combined with the pulse mappings for each segment to generate suitable strengths for each beat. The first beat of the bar (the downbeat) is where phrases usually begin, so it provides anchors for the succeeding agents. As the tempo agent reduces the landmarks from fames to beats, this agent is much more suited to a genetic algorithm, and so a form of pattern matching is used to provide fitness measures.

The chromosome of the pulse agent genetic algorithm consists of a string of numbers, with each representing the strength of the beat at that point. Hence [1,0] suggests the first bar of a standard 2/4 beat. There are enough genes in each chromosome to accommodate the whole of the section, which can be quite large for higher tempos (a 5 minute segment at 240 bpm would have 1,200 beats). However, the tempo is usually closer to 120pm and shorter note lengths are used to suggest the faster speed.

Three operators are used in this agent: crossover, mutation, and sequence. Crossover operates in the traditional manner, with a range of genes being swapped between two

FIGURE 8.3: Output from the Pulse Agent

chromosomes. Mutation is also based on the standard operator, with a random gene changed to a random value. The sequence operator, however, is unique to this system, and is used to provide repetition. A range is selected within the chromosome, from which a non-unary factor is chosen. A string of genes of this length is then repeated throughout the section, with the genes selected from the start of the range. Repetition is fundamental to the pulse, so an operator to promote it is ideal in this situation.

The fitness function of the pulse agent is customized to guide the GA towards a suitable solution. From left to right, the beat options suggested by the composer representation are compared to a substring that is as long as the segment. For every gene that matches, $1/l$ is added to that option's score (where l is the length of the substring). If one or more matches are exactly correct, the most highly weighted match is chosen and the weight added to the score. Otherwise, the closest match is returned, and the score multiplied by the weight is added. Hence, if only half the string is matched, the final score is halved. An example of the results of this agent is shown in Figure 8.3, where two options were available to the genetic algorithm: [1, 0, 0.25, 0.75] and [1, 0, 1, 0], both with equal weightings of 1. Bars with alternating accented notes (bars 1, 2, and 3) conform to the [1, 0, 1, 0] sequence, while the final bar, with its crescendoing final two notes, conforms to the [1, 0, 0.25, 0.75] sequence. The produced pulse has a fitness of 60 for a chromosome of length 16.

On completion, the pulse agent alters the 'strength' attribute of the markers provided by the beat agent to reflect the final pulse genes. These are incorporated into the landmark file and passed on to the next agent in the graph.

8.4 Key Agent

Chromosome Encoding: Each gene contains a unique string refering to the key at that point.

Genetic Operators: Crossover (1/5 probability), mutation (1/8 probability).

Fitness Function: Stepping through a graph with keys on the vertices, each edge score is added if one is present.

Population Size: 1000

Selection Approach: Threshold selection

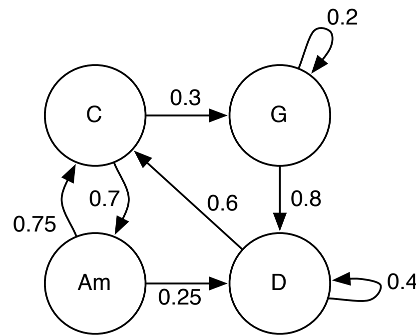


FIGURE 8.4: An example key graph. Note the probabilities of transitioning from one key to another.

The key agent is the first in the framework to use a graph-based approach. While rhythm and pulse are simpler to specify as short cases, key changes are easier to represent as the probability of moving from one key to another. A scoring approach was created using this graph technique, which is at the core of the key agent fitness function.

The first step in the key agent is to build up the key graph for the segment that is being evolved. A directed, weighted, graph is used, with each node being a key and each edge representing a key change (see Figure 8.4). The key change landmark is read from the provided landmark parameter, and a graph is constructed with the correct weights on each edge.

The chromosome for the key agent is simple, with each gene representing one of the keys available. For example, 'Cmaj' could be represented by 0, 'Dmaj' as 1, and so on. It is not necessary to encode more information into the gene, as the root and scale may be obtained from the landmark file when required. The chromosome length is set by the number of key changes required in the block, so if 10 key change landmarks are indicated, 10 genes are present in the chromosome. The traditional two operators (crossover and mutation) are used for this agent, with mutation setting the gene to a random key.

To make use of the graph data structure, a custom genetic algorithm was written. This moves by step through the chromosome and progresses through the graph from node to node. If a weight is present on an edge, this score is added to the final total. As each edge is a probability, dividing the total by the sequence length provides the probability of the chromosome matching the requirements. On completion, the chosen keys are inserted into the landmark file as key markers, with a 'ref' attribute denoting which key should be used.

8.5 Chord Agent

Chromosome Encoding: Each gene contains a unique string refering to the chord at that point.

Genetic Operators: Crossover (1/5 probability), mutation (1/8 probability).

Fitness Function: Stepping through a graph with chords on the vertices, each edge score is added if one is present. If the edge is a cadence, a score is only added if it is at the end of the chromosome.

Population Size: 1000

Selection Approach: Threshold selection

As with the key agent, a graph-based technique is used to develop the underlying chords for the score. These are essential to the melody evolution later in the process, as notes in the chord have priority over non-chordal notes. As with keys, there are some predefined rules for chord progression, which can be handled by the graph approach.

Being similar in operation to the key agent, the chord agent first builds up a chord graph to represent the progressions available in the system. In this instance, the nodes in the graph refer to the unique names defined in the composer representation, such as 'I' or 'V7', and these are encoded as strings in the chromosome. The nodes are linked by the defined chord progressions, and an extra flag is present in the edge definition to specify whether the link can be a 'cadence' (an edge linking the final two chords of a segment). The same operators are used as in the key agent, but in this case the mutation operator selects a random chord.

The fitness function uses the same technique as the key agent, but with one difference. If the edge between two chords is defined as a cadence, it only receives a score if it is at the end of the chromosome (or if the edge is defined twice – once as a cadence, once as a basic progression). This aims to finish the piece on a suitable ending. Chords are generated for landmarks containing strong beat markers, and these are inserted as markers in the appropriate landmarks. These are again similar to the key agent, with the ref attribute indicating the appropriate chord.

8.6 Instrumentation Agent

The simplest agent in the framework, the instrumentation agent selects the number of parts in the score and which instruments should be assigned to each part. Each instrument in the system can be weighted, and this mapping is passed as a parameter to the agent. To choose a set of instruments, the instrument weightings provided by the

Instrument	Weighting
Piano	6
Violin	5
Flute	5
Oboe	4
Guitar	2
Trumpet	1

TABLE 8.1: A simple instrument/weighting mapping for a segment.

landmark representation are accumulated. From this value the mean is found, and the instruments with weightings greater than this are chosen for the score. For example, given the instrument mapping in Table 8.1, $\overline{x} = 3.83$, so piano, violin, flute, and oboe would be selected for the segment as their weightings are all greater than this value. Note that this is a very simple approach, and future work could consider common instrument groupings (such as string quartets)..

8.7 Rhythm Agent

Chromosome Encoding: A floating point value per gene representing the note length and a Boolean denoting whether the note is a rest.

Genetic Operators: Crossover (1/3 probability), mutation (1/30 probability - customized to handle the rest Boolean).

Fitness Function: Similar to the Pulse Agent, except only the portion of the chromosome within the pulse timescale is considered. Also, the weightings are multiplied by the pulse strength.

Population Size: 500

Selection Approach: Tournament selection

The next agent in the agent graph is the rhythm agent, which makes use of the output of the pulse agent to determine the note lengths to be used in the segment. Again, a genetic algorithm is used to produce the final rhythm, but in this case the rhythm is created for each instrument in the score. As such, this is also the first case in which the process may be parallelized by part, and hence is an ideal candidate for the agent graph approach.

To allow for multiple instruments, a multiple pass approach is carried out to build up state information structures. This first goes through each segment, locating changes in the instrumentation state, and storing these with their positions in the music. A second pass then collates pulse and rhythm information for each instrumentation block. Finally,

Figure 8.5: Output from the Rhythm Agent

a chromosome is evolved for each instrument, with the pulse and rhythm parameters passed to the individual genes and the fitness function.

The chromosome of the rhythm agent is closely tied to the format used in the composer representation. As was described previously, each note length is assigned a decimal value (4 for a semibreve, 2 for a minim, and so on). The rhythm gene contains both this value and a Boolean to denote whether the note should be sounded or treated as a rest. The length of the chromosome is difficult to determine, as a rhythm that is entirely semiquavers would be 8 times longer than a rhythm comprised of minims. As such, a chromosome is reserved that can hold enough semiquavers to cover the segment. Unfortunately this can become very large, so a maximum size can be specified to reduce the overhead of the algorithm. To improve efficiency, only the section of the chromosome that is of a suitable duration is scored.

The rhythm agent uses a similar set of operators to the pulse agent, with the mutation altered to handle the 'rest' parameter of the rhythm gene. The fitness function is also based on the same approach of weighted pattern matching, although there are a few pertinent differences:

1. The duration of the chromosome is checked during the evaluation stage, and only the portion that is within the timescale of the pulse information is examined. This prevents the need to examine all of the genes within the chromosome.
2. The pattern weightings are multiplied by the value of the pulse at the start of the pattern. For example, if a pulse has a strength of 0.5 and the pattern in that location is weighted as 1.5, 0.75 is added to the final score. This guides the algorithm towards preferring rhythms that begin on strong beats of the bar, while allowing for some variation.

8.8 Melody Agent

Chromosome Encoding: Each gene contains a pitch value that references a note within the scale.

Genetic Operators: Crossover (1/3 probability), mutation (1/30 probability).

Figure 8.6: Output from the Melody Agent

Fitness Function: 1. Pitch probabilities and pulse probabilities are multiplied and accumulated to ensure low-probability notes exist on low-probability beats. 2. Using the pitch graph, edge probabilities are accumulated (as with the Key Agent).

Population Size: 500

Selection Approach: Tournament selection

The melody agent takes parameters from several preceding agents to build up the necessary pitch graphs. The Key Agent provides the scale information (essentially the notes that are available), the Chord Agent weights this with the notes that should be used most often, the Instrumentation Agent ensures notes can be played, and the Pulse Agent indicates the beat strengths to determine whether a non-chord note can be placed, and the Rhythm Agent markers give the positions of the notes in the music.

As with key and chord, a graph-based approach is used for generation. The nodes are populated using the scales, pruned if they are not playable with the current instrument, and then weighted via the chord modifiers. Finally, the edges are weighted based on the melody modifiers that are passed to the agent.

A GA-approach is utilised again, with each chromosome containing a sequence of pitches. The graph is utilised in the fitness function, with a two-stage scoring process. First, the nodes are handled. For each note marker, the pitch probability and pulse strength are used to accumulate the initial score. This has two cases - if the pitch probability and pulse strength are high (greater than 0.5) the two are multiplied. If the pitch probability and pulse strength are low (less than or equal to 0.5) the pulse strength is inverted and the two are again multiplied. This allows low-probability notes on low-priority beats, and vice versa.

The second stage handles the edges. Starting with the initial note gene, edge probabilities are simply accumulated. If no edge is available, the edge is skipped and a new node is chosen. As such, a melody that contains intervals that are weighted in the composer mesh will be more highly rated than one that does not. Figure 8.6 shows a melody produced by the melody agent when restricted by only the key parameters (in this case, C major).

8.9 Summary

The decomposition of composing into its component tasks is central to SBS and, as has been shown, this allows for a powerful amount of specialisation in each composing agent. The agent graph is simple at present but may be easily extended to allow for further agents and algorithms, such as motif or dynamics. Due to the simplicity of the landmark file, it is also straightforward to alter the underlying algorithms while ensuring the inputs and outputs remain the same. For example, the pulse agent could be replaced with a grammatical approach with no change needed to previous stages. To summarise, the agent-based approach to composition illustrated here allows for a highly flexible method - both in its extensibility and expandability.

Chapter 9

A Case Study

To demonstrate the capabilities of the SBS system, a 1378 frame scene from *Total Recall*[1] was annotated such that the script and associated characters could be represented by the OntoMedia framework, and this result mapped to a landmark file for musical composition. This chapter steps through this process, from the initial script annotation[2], to the OntoMedia representation, composer mapping, landmark file, and eventual agent outputs.

9.1 Screenplay Annotation

The first step in the annotation of the chosen scene was the marking up of the screenplay. This served two purposes: First, it gave an indication as to the characters present in the scene, and secondly it provided information regarding the key events in the scene.

As described in Chapter 4, a markup language called SiX was designed for this purpose as part of the project. This was used to annotate the scene, and Listing 9.1 shows a portion of the 25 line annotated version. In this form, it is straightforward to locate the pertinent information. For example, the location is specified in the <sc:location> tags, Lori and Quaid are the two primary characters, and there are three key directions that can be tied to OntoMedia events.

In this case study we do not take dialogue into consideration, but it is also apparent from the SiX markup where these Action classes should be. This information could be used as an aid to voice identification tools, especially in conjunction with the links to the actors which OntoMedia provides.

Once the screenplay is annotated, it is possible to link directly from OntoMedia to the script representation. This is achieved with the use of ScriptItem objects, which extend

[1]http://imdb.com/title/tt0100802/
[2]http://ecs.soton.ac.uk/ moj/script/totalrecall.xml

```
<?xml version="1.0" encoding="iso-8859-1" standalone="no" ?>
<sc:script xmlns="http://www.w3.org/1999/xhtml"
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:sc="http://mikesroom.org/script">

  <sc:info>
    <dc:title>Total Recall</dc:title>
  </sc:info>

  <sc:location time="day" pos="int">HILTON - CORRIDOR/SERVICE ELEVATOR - 6TH FL.
    </sc:location>
  <sc:dialogue speaker="Lori">Doug...you wouldn&apos;t hurt me, would you, honey?
    </sc:dialogue>
  <sc:direction>She sees his expression.</sc:direction>
  <sc:dialogue speaker="Lori" ctd="1">Sweetheart, be reasonable...We&apos;re
    married.</sc:dialogue>
  <sc:direction>Lori stealthily reaches behind her back for a concealed gun and
    pulls it on him.</sc:direction>
  <sc:direction>Quaid shoots Lori in the forehead, leaving a clean, small hole
    between her eyes.</sc:direction>
  <sc:dialogue speaker="Quaid" paren="rising, to Melina">Consider that a divorce.
    </sc:dialogue>
</sc:script>
```

Listing 9.1: A section from *Total Recall* in SiX format

```
<omsi:ScriptItem rdf:ID="Scene1">
  <omsi:uri>http://ecs.soton.ac.uk/~moj/script/totalrecall.xml</omsi:uri>
  <omsi:has-subregion rdf:resource="#Loc1" />
  <omsi:has-subregion rdf:resource="#Act1" />
</omsi:ScriptItem>

<omsi:ScriptItem rdf:ID="Loc1">
  <rdfs:label>HILTON - CORRIDOR/SERVICE ELEVATOR - 6TH FL.</rdfs:label>
  <omsi:uri>http://ecs.soton.ac.uk/~moj/script/totalrecall.xml#xpointer(element(
    scene1/1))</omsi:uri>
  <omsi:has-expression rdf:resource="#Corridor1" />
</omsi:ScriptItem>
```

Listing 9.2: Binding an OntoMedia representation to a SiX screenplay resource

the MediaItem class discussed in Chapter 5. To provide a flexible means of referencing the marked screenplay, xpointers are stored in attributes of the objects, so references may be specific (pointing to an element with a given id) or relative (pointing to the Nth element of a file) and a script may be held in multiple files. Furthermore, as with other MediaItems, a ScriptItem may have subregions. As such, it is possible to have one ScriptItem representing a scene, and that containing other ScriptItems representing individual script elements.

To tie these items to OntoMedia Expressions, the has-expression property is employed. Listing 9.2 shows two ScriptItems, one covering the whole screenplay file, and a second tying a Location to its script representation. It is, of course, possible to bind from a ScriptItem to an occurrence, which is essential for the linking of action or dialogue to the corresponding script information.

```
<oms:Building rdf:ID="Hilton"></oms:Building>

<oms:Corridor rdf:ID="Corridor1">
<oms:is-part-of rdf:resource="#Floor6" />
</oms:Corridor>

<oms:Lift rdf:ID="ServiceLift">
  <oms:permits-access-to rdf:resource="#Floor6" />
  <oms:is-visible-from rdf:resource="#Corridor1" />
  <oms:permits-viewing-of rdf:resource="#Corridor1" />
</oms:Lift>

<oms:Floor rdf:ID="Floor6">
  <oms:is-part-of rdf:resource="#Hilton" />
</oms:Floor>
```

Listing 9.3: Defining the location of a scene

9.2 Location Annotation

As was seen in the previous listing, it is possible to bind from a screenplay's location description to a Location object within OntoMedia. The Signage Location ontology[3], upon which the OntoMedia location ontology is based, provides for building description, and this is extended further to cater for exterior locations. The ontology includes linking properties to provide the ability to specify how one location relates to another. For example, it is possible to say that a room is on the fourth flow of a building. In this example we want to represent the 6th floor of the Hilton hotel, and more specifically the corridor and service elevator on this level.

Listing 9.3 shows the description of this information in OntoMedia. First the Hilton is defined as a Building instance, and Floor6 is described as being part of the Hilton building. Next, the corridor in which the scene occurs is placed as part of this floor. Finally, the service lift is specified. This is slightly more involved, and is described as allowing access to Floor 6 (as well as the others, but these are not featured in the film), as being visible from the corridor, and as allowing an occupant to see the corridor. This information is ideal for users who may wish to verify continuity information ('Can this character really see this character?').

9.3 Character Annotation

To ease the annotation of the (otherwise complex) character instances, the Meditate application was designed to act as an easy-to-use interface between the user and the RDF. By taking the options available to the user directly from the OWL ontology definition and exporting the created entities in RDF, Meditate removes the complexity of editing RDF files.

[3] Millard et al. [2004]

Meditate provides the ability to mark up a large amount of information regarding characters, including the links between them. Figure 9.1 shows the Meditate display for the Douglas Quaid character in *Total Recall*. As well as basic information about the character, such as names and gender, links to other related entities are also visible - in this case the Being representing Arnold Schwarzenegger and the Character representing the character "Douglas Quail" from the Philip K. Dick short story "We Can Remember it for you Wholesale"[4] upon which *Total Recall* was very loosely based.

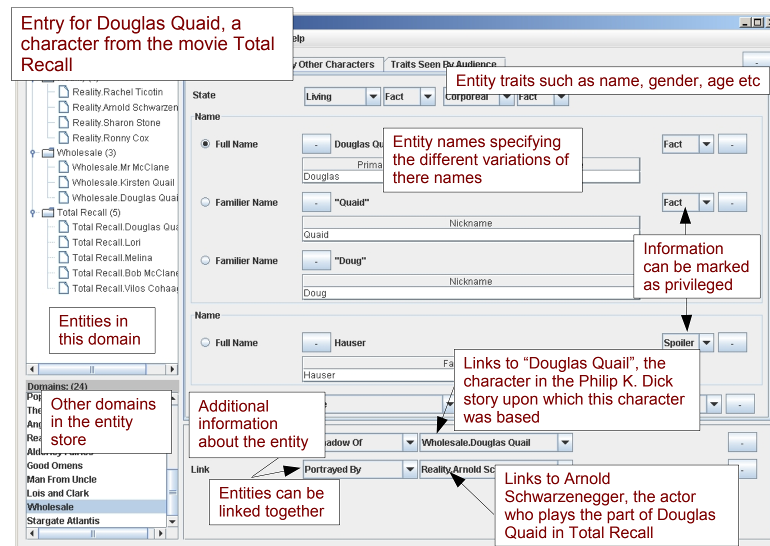


FIGURE 9.1: Meditate: Entry for Character "Douglas Quaid"

While Meditate allows for local saving and loading of RDF files, it is also able to retrieve and deposit RDF descriptions into an 'entity store'. This provides simple version control, as well as multiuser support, so it is possible to edit the item as a collaboration between several people. The RDF generated from the entry in Figure 9.1 can be seen in Listing 9.4.

9.4 Event Annotation

OntoMedia contains three elements which are essential for the content within a medium: Timeline, Event, and Occurrence. All of these are necessary to fully describe a sequence of events, although not all events need actually occur (a character may *want* an event to occur, but it may not actually exist on a timeline). A medium may also have several timelines, where one timeline may be another character's view, or consist of events which occur in a dream.

In the case of the *Total Recall* segment, we are annotating a fight sequence (see Figure 9.2). Douglas Quaid, the male protagonist, is being dragged unconscious to an elevator

[4] Dick [1969]

```
<rdf:Description rdf:about="#Total_Recall_Douglas_Quaid_Character_102">
  <rdfs:label>Total Recall.Douglas Quaid</rdfs:label>
  <rdf:type>
    <owl:Class rdf:about="&being;Character" />
  </rdf:type>
  <ontomedia:exists_in rdf:resource="#Total_Recall_Context_101"/>
  <!-- A subclass of State-Of-Being, the Alive trait states that the entity is
    living -->
  <trait:has-trait rdf:resource="#_new_entity_34__Alive_Fact_103"/>
  <!-- A subclass of State-Of-Form, the Corporeal trait states that the entity is
     corporeal -->
  <trait:has-trait rdf:resource="#_new_entity_5__Corporeal_Fact_104"/>
  <!-- These Name references describe the name of the character at this point. As
     he has two (Douglas Quaid and Hauser) one is a spoiler. -->
  <trait:has-trait rdf:resource="#_new_entity_45__Name_Fact_105"/>
  <trait:has-trait rdf:resource="#_new_entity_47__Name_Spoiler_122"/>
  <!-- A subclass of Gender, the Male trait states that the entity is male -->
  <trait:has-trait rdf:resource="#_new_entity_24__Male_Fact_127"/>
  <ontomedia:is-shadow-of rdf:resource="#Wholesale_Douglas_Quail_Character_1"/>
  <being:portrayed-by rdf:resource="#Reality_Arnold_Schwarzenegger_Being_58"/>
</rdf:Description>
```

Listing 9.4: Meditate RDF for Douglas Quaid

Figure 9.2: A portion of the occurrences in the Total Recall timeline.

by his captor, Lori. Once they reach the elevator, Quaid's ex-lover (and the female protagonist) Melina arrives in the elevator, and the fight commences. This involves instances where Lori has the upper hand, where Melina has the upper hand, and, finally, where Quaid takes control of the situation and kills Lori. There are also two instances where both Lori and Melina lose items - one where Lori's gun is kicked away, and one where Melina's knife is shot away.

As such, four subclasses of Event are utilised - Action, Transformation, Gain, and Loss. In this instance, Action is used for a scene in which little occurs plotwise, but there is action (a fight in this case); Loss/Gain are used to represent a character losing or gaining an item; and Transformation to represent the alteration of a character's state. Listing 9.5 shows all of these cases. It can be easily inferred that Lori has the upper hand in the first event, as she is specified as the subject entity, and that the gun kicked away by Lori in Act3Eve2 is retrieved by Quaid in Act6Eve. Finally, the content of Act12Eve2 makes it clear that the character Lori changes state from Alive to Dead.

Once the events are declared, it is a simple matter to create the timeline and add the required occurrences. Occurrences are very straightforward: they specify the position of

```
<ome:Action rdf:ID="Act3Eve">
  <ome:has-occurrence rdf:resource="#Act3Occ" />
  <ome:has-subject-entity rdf:resource="&base;Total_Recall_Lori_Character_145" />
  <ome:has-object-entity rdf:resource="&base;Total_Recall_Melina_Character_154"
     />
  <ome:has-location rdf:resource="#Corridor1" />
</ome:Action>

<ome:Loss rdf:ID="Act3Eve2">
  <ome:has-occurrence rdf:resource="#Act3Occ" />
  <ome:has-subject-entity rdf:resource="&base;Total_Recall_Melina_Character_154"
     />
  <ome:has-object-entity rdf:resource="&base;Total_Recall_Gun1" />
  <ome:has-location rdf:resource="#Corridor1" />
</ome:Loss>

<ome:Gain rdf:ID="Act6Eve">
  <ome:has-occurrence rdf:resource="#Act6Occ" />
  <ome:has-subject-entity rdf:resource="&base;
    Total_Recall_Douglas_Quaid_Character_102" />
  <ome:has-object-entity rdf:resource="&base;Total_Recall_Gun1" />
  <ome:has-location rdf:resource="#Corridor1" />
</ome:Gain>

<ome:Transformation rdf:ID="Act12Eve2">
  <ome:has-occurrence rdf:resource="#Act12Occ" />
  <ome:has-subject-entity rdf:resource="&base;Total_Recall_Lori_Character_145" />
  <ome:from><omt:Alive /></ome:from>
  <ome:to><omt:Dead /></ome:to>
  <ome:has-location rdf:resource="#Corridor1" />
</ome:Transformation>
```

Listing 9.5: Defining the events within a scene

```
<ome:Timeline rdf:ID="Timeline" />

<ome:Occurrence rdf:ID="Act1Occ">
  <ome:involves rdf:resource="&base;Total_Recall_Douglas_Quaid_Character_102" />
  <ome:involves rdf:resource="&base;Total_Recall_Lori_Character_145" />
  <ome:precedes rdf:resource="#Act2Occ" />
  <ome:timeline-ref rdf:resource="#Timeline" />
</ome:Occurrence>
```

Listing 9.6: Defining the occurrences of events

the events on the timeline relative to the other occurrences. This also allows for events to be used multiple times, or for events to coincide. It is evident from the previous listing that Act3Eve and Act3Eve2 occur at the same time, but we are not able to infer any information regarding the ordering at this point. Listing 9.6 demonstrates how the Timeline and Occurrences are created. In this case we also use the Occurrence to denote other characters in the scene, who may not be part of the primary events.

9.5 Querying the OntoMedia Representation

To test the flexibility of OntoMedia, the annotated scene from *Total Recall* was imported into a Sesame triple store. The following are a few examples of the RDQL queries and the results which were obtained.

9.5.1 All People

This query returns the primary and family names of all beings (actors in this case). Names are stored in OntoMedia using name components, so it is possible to store multiple names for the same being, with titles and nicknames, while allowing for foreign word orderings.

```
SELECT  ?primaryname, ?familyname
WHERE   (?being rdf:type being:Being)
        (?being omt:has-trait ?name)
        (?name rdf:type omt:Name)
        (?name omt:has-name ?fullname)
        (?fullname rdf:type name:Full)
        (?fullname name:has-order ?order)
        (?order rdfs:member ?namepart)
        (?namepart rdf:type name:Primary-Name)
        (?namepart name:has-name-component ?primaryname)
        (?order rdfs:member ?namepart2)
        (?namepart2 rdf:type name:Family-Name)
        (?namepart2 name:has-name-component ?familyname)
using
        rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
        rdfs for <http://www.w3.org/2000/01/rdf-schema#>,
        being for <http://ontomedia.ecs.soton.ac.uk/ontologies/ext/common/being#>
    ,
        omt for <http://ontomedia.ecs.soton.ac.uk/ontologies/ext/common/trait#>,
        name for <http://ontomedia.ecs.soton.ac.uk/ontologies/misc/name#>

Result:

"Ronny" "Cox"
"Sharon" "Stone"
"Arnold" "Schwarzenegger"
"Rachel" "Ticotin"
```

9.5.2 Linking Actors to Characters

Each character in a media item may have an accompanying actor. In this example a list of actor references is retrieved, together with associated characters.

```
SELECT  ?being, ?label
WHERE   (?being rdf:type being:Being)
        (?being being:portrays ?character)
        (?character rdfs:label ?label)
using
        rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
        rdfs for <http://www.w3.org/2000/01/rdf-schema#>,
        being for <http://ontomedia.ecs.soton.ac.uk/ontologies/ext/common/being#>


Result:

TotalRecall#Reality_Ronny_Cox_Being_211 "Total Recall.Vilos Cohaagen"
TotalRecall#Reality_Sharon_Stone_Being_197 "Total Recall.Lori"
```

```
TotalRecall#Reality_Arnold_Schwarzenegger_Being_58 "Total Recall.Douglas Quaid"
TotalRecall#Reality_Rachel_Ticotin_Being_183 "Total Recall.Melina"
```

9.5.3 Locating Specific Events

Finally, this example focuses closer to the storyline of *Total Recall*. The query requests the script items and events in which Lori (as the subject entity) loses an item.

```
SELECT  ?uri
WHERE   (?item rdf:type omsi:ScriptItem)
        (?item omsi:uri ?uri)
        (?item omsi:has-expression ?expr)
        (?expr rdf:type ome:Occurrence)
        (?event ome:has-occurrence ?expr)
        (?event rdf:type ome:Loss)
        (?event ome:has-subject-entity ?char)
        (?char rdfs:label "Total Recall.Lori")
using
        rdf for <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
        rdfs for <http://www.w3.org/2000/01/rdf-schema#>,
        ome for <http://ontomedia.ecs.soton.ac.uk/ontologies/core/expression#>,
        trait for <http://ontomedia.ecs.soton.ac.uk/ontologies/ext/common/trait#>
    ,
        omsi for <http://ontomedia.ecs.soton.ac.uk/ontologies/ext/media#>

Result:

"http://ecs.soton.ac.uk/~moj/script/totalrecall.xml#xpointer(element(scene1/9))"
           #Act8Eve2
```

9.6 Creating the Landmark Representation

Once the annotated *Total Recall* was imported into the OntoMedia triplestore, it was possible to create the necessary files for the construction of the Landmark Representation. As input, this stage simply requires a pointer into the triple-store and a composer representation (see Chapter 6). A relative straightforward mapping was used in this instance:

9.6.1 Location: Mars

Space movies often have similar characteristics to their music in order to depict the enormity of the environment. Mars, both in real life and as it is shown in *Total Recall*, is a barren, inhospitable, planet. As such, strings and woodwind playing in a smooth, chromatic, style can portray the winds over the surface as well as providing an unsettling backdrop. To enhance the vastness of the location, wide intervals may be used - in this

case in the brass part. This is partly due to the ease of playing fifths on brass instruments (for example, in Star Wars or 2001: A Space Odyssey), but also as they contrast well with the close intervals of the strings and woodwind beneath.

Note that the scene taken for this case study, while located on Mars, is located in a corridor of a hotel. However, the Martian theme is still important, so it is allowed to influence the result.

```
<map>
  <query>
    (event ome:has-location ?loc)
    (?loc rdfs:label "Total Recall.Mars")
  </query>
  <promote>
    <param id="wind_effect" />
    <param id="brass_fifths" />
    <param id="woodwind" />
    <param id="brass" />
    <param id="strings" />
  </promote>
</map>

<modifiers name="wind_effect" affects="woodwind,strings">
<modifier type="melody">
  <sequenced>
    <note number="1" />
    <note number="2" />
  </sequence>
  <sequence>
    <note number="2" />
    <note number="3" />
  </sequence>
  ...
</modifier>
</modifiers>

<modifiers name="brass_fifths" affects="brass">
<modifier type="melody">
  <sequence>
    <note number="1" />
    <note number="5" />
  </sequence>
  <sequence>
    <note number="5" />
    <note number="1" />
  </sequence>
</modifier>
</modifiers>
```

Listing 9.7: The composer representation for Mars.

9.6.2 Characters: Quaid and Melina

Quaid is the male lead of *Total Recall*, and has three tasks in the film - to reclaim his previous memories, to eliminate the tyrannical ‘adminstrator’ of Mars, and, as a result, provide oxygen for the inhabitants of the planet. In the film, Quaid is very much the archetypal action hero, so the music should reflect the strength and determination of

the character. To achieve this, the composer mapping promotes a major key, dotted rhythms, a preference towards strings and brass for instrumentation.

The female lead, Melina, is Quaid's love interest in the movie. She is part of a resistance movement working to overthrow the administrator and has a very similar character to Quaid. As such, the composer mapping is much the same, but with woodwind in place of the brass to provide a contrast.

```
<map>
  <query>
    (?event ome:has-subject-entity ?char)
    (?char rdfs:label "Total Recall.Quaid")
  </query>
  <promote>
    <param id="major_key" />
    <param id="dotted_rhythm" />
    <param id="brass" />
    <param id="strings" />
</promote>
</map>

<map>
  <query>
    (?event ome:has-subject-entity ?char)
    (?char rdfs:label "Total Recall.Melina")
  </query>
  <promote>
    <param id="major_key" />
    <param id="dotted_rhythm" />
    <param id="strings" />
    <param id="woodwind" />
</promote>
</map>

<modifiers name="major_key">
  <modifier type="key" mode="scale">
    <pitch pos="1" />
    <pitch pos="3" />
    <pitch pos="5" />
    <pitch pos="6" />
    <pitch pos="8" />
    <pitch pos="10" />
    <pitch pos="12" />
  </modifier>
</modifiers>

<modifiers name="dotted_quaver">
  <modifier type="rhythm">
    <sequence>
      <note length="0.75" />
      <note length="0.25" />
    </sequence>
  </modifier>
</modifiers>
```

Listing 9.8: The composer representation for Quaid and Melina.

9.6.3 Character: Lori

At the start of *Total Recall*, the audience is introduced to her as Quaid's wife. However, as the film progresses it is revealed that she is in fact working for the administrator to

prevent Quaid from achieving his goals. In the scene, she attempts to prevent Melina and Quaid from escaping. As such, she is the villain of the piece, and the music suggests this. Lori's modifiers include a minor key signature, together with fast string technique to allude to a dangerous tension.

```
<map>
  <query>
    (?event ome:has-subject-entity ?char)
    (?char rdfs:label "Total Recall.Lori")
  </query>
  <promote>
    <param id="minor_key" />
    <param id="short_rhythm" />
    <param id="strings" />
</promote>
</map>

<modifiers name="minor_key">
  <modifier type="key" mode="scale">
    <pitch pos="1" />
    <pitch pos="3" />
    <pitch pos="4" />
    <pitch pos="6" />
    <pitch pos="8" />
    <pitch pos="9" />
    <pitch pos="12" />
  </modifier>
</modifiers>

<modifiers name="short_rhythm">
  <modifier type="rhythm">
    <sequence>
      <note length="0.5" />
      <note length="0.5" />
    </sequence>
  </modifier>
</modifiers>
```

LISTING 9.9: The composer representation for Lori.

9.6.4 Interactions

The annotated scene involves three key event types: The alteration of advantage in a fight (between Melina, Lori, and Quaid), the gain/loss of a weapon (be it a gun or a knife), and the eventual death of Lori. The first case is automatically handled by the system, as the subject entity is the character with the advantage in the fight. The weapon transfer has two variants - a major theme is promoted if a 'good character' has the weapon, and a minor theme otherwise. Finally, a less chromatic segment is introduced involving strings and woodwind upon the death of Lori. Listing 9.10 shows the mappings for the second case - denoting the gain of an item.

```
<map>
  <query>
        (?expr rdf:type ome:Occurrence)
        (?event ome:has-occurrence ?expr)
        (?event rdf:type ome:Gain)
        (?event ome:has-subject-entity ?char)
        (?char rdfs:label "Total Recall.Lori")
        (?event ome:has-object-entity ?item)
        (?item rdfs:label "Total Recall.Gun1")
  </query>
  <promote>
    <param id="minor_key" />
</promote>
</map>

<map>
  <query>
        (?expr rdf:type ome:Occurrence)
        (?event ome:has-occurrence ?expr)
        (?event rdf:type ome:Gain)
        (?event ome:has-subject-entity ?char)
        (?char rdfs:label "Total Recall.Quaid")
        (?event ome:has-object-entity ?item)
        (?item rdfs:label "Total Recall.Gun1")
  </query>
  <promote>
    <param id="major_key" />
</promote>
</map>
```

Listing 9.10: A composer representation capable of altering key when guns are lost.

9.6.5 Generating the Landmark File

Once the mapping file is complete, it is passed through the translator, described in Chapter 6. This creates probability values for the different modifiers based on those promoted in the mapping, and produces the landmark file. Listing 9.11 shows the first landmark of the segment. In this instance both Lori and Quaid are present, so we see their modifiers in place. Furthermore, the scene is taking place on Mars, so its parameters are also brought through. Lori is the dominant character in this instance, so her modifiers (a minor key, short rhythms, etc) take precedence. Note that some sensible default values are also brought in via the translation, with a set of relative keys, and some chord progressions.

9.7 Composing The Music

From the landmark file, the composition process is passed onto the Light Agent Framework and the connected musical agents. As was discussed in Chapter 8, this leads through a sequence of agents capable of handling tempo, pulse, key, chord, instrumentation, rhythm, and melody. At each stage the landmark file is updated, and MusicXML containing the amended piece is passed through to the next agent. To show this process, the outputs from each stage are given in this section.

```
&lt;segment from="0" to="1378"&gt;
  <landmark at="0">
    <modifier probability="0.4" ref="dotted_quaver" type="rhythm"/>
    <modifier probability="0.25" ref="crotchet" type="rhythm"/>
    <modifier probability="0.25" ref="major_key" type="key"/>
    <modifier probability="0.75" ref="minor_key" type="key"/>
    <modifier probability="0.1" ref="Am" type="key"/>
    <modifier probability="0.1" ref="C" type="key"/>
    <!-- etc -->
    <modifier probability="1.0" ref="relative" type="key"/>
    <modifier probability="0.5" ref="march" type="pulse"/>
    <modifier probability="0.5" ref="slow_march" type="pulse"/>
    <modifier probability="0.5" ref="wind_effect" type="melody"/>
    <modifier probability="0.5" ref="brass_fifths" type="melody"/>
    <modifier probability="0.5" ref="strings" type="instrumentation"/>
    <modifier probability="0.17" ref="woodwind" type="instrumentation"/>
    <modifier probability="0.33" ref="brass" type="instrumentation"/>
    <modifier probability="0.25" ref="II-V" type="chord"/>
    <modifier probability="0.25" ref="V-I" type="chord"/>
    <!-- etc -->
    <modifier probability="0.25" ref="IV-III-IV" type="chord"/>
    <modifier probability="0.25" ref="V-VI" type="chord"/>
  </landmark>
&lt;/segment&gt;
```

LISTING 9.11: The landmark generated for the first shot of the scene.

9.7.1 Tempo Agent

As there are no tempo markings present in the modifiers, the tempo agent uses landmark locations to predict a plausible tempo. In this case, it has selected an Andante tempo (88bpm). Listing 9.12 shows the resultant landmark file, and Listing 9.13 shows the MusicXML representation. Note the markers placed in the landmark XML with their initially zeroed strength values.

9.7.2 Pulse Agent

Following on from the tempo agent, the pulse agent inserts appropriate strengths for the beats. These also indicate where bar lines should be placed. Two different pulses can be seen in Listing 9.14, with a metronomic tempo over the first four beats and a more march-style sequence over the final four. As a result, the MusicXML (see Listing 9.15) contains two bars (or measures) in place for this fragment.

9.7.3 Key Agent

Once the temporal aspects of the segment have been decided, the harmonic aspects may be chosen. The first step of this is the key agent, which inserts a key signature in the first landmark of the segment. In this case D minor was selected, based on the preference of a minor key. The MusicXML form represents this in terms of the cycle of fifths, with negative and positive values introducing more flats and more sharps respectively. D minor has a single flat, so -1 is used.

```
<!-- ... -->
&lt;segment from="0" to="1378"&gt;
  <landmark at="0">
    <-- ... -->
    <marker strength="0.0" type="beat"/>
  </landmark>
  <landmark at="17">
    <marker strength="0.0" type="beat"/>
  </landmark>
  <landmark at="34">
    <marker strength="0.0" type="beat"/>
  </landmark>
  <landmark at="34">
    <marker strength="0.0" type="beat"/>
  </landmark>
  <landmark at="51">
    <marker strength="0.0" type="beat"/>
  </landmark>
  <landmark at="68">
    <marker strength="0.0" type="beat"/>
  </landmark>
  <landmark at="86">
    <marker strength="0.0" type="beat"/>
  </landmark>
  <landmark at="92"/>
  </landmark>
  <!-- ... -->
&lt;/segment&gt;
```

Listing 9.12: The tempo landmarks generated for the first shot of the scene.

```
<score-partwise>
  <movement-title>Untitled</movement-title>
  <part-list>
    <score-part id="P1" />
  </part-list>

  <part id="P1">
    <measure number="1">
      <direction placement="above">
        <direction-type>
          <words font-weight="bold" relative-x="-40">Andante</words>
        </direction-type>
        <sound tempo="88"/>
      </direction>
    </measure>
  </part>
</score-partwise>
```

Listing 9.13: The MusicXML tempo generated for the first shot of the scene.

9.7.4 Chord Agent

The chord agent varies slightly from the others in that it does not alter the MusicXML file. The chord information is not necessary for the musical notation, while it is essential to the melody agent. While the full chord sequence is too long to show here in XML form, the entire sequence for this section was: II V II V II V II V V VI II V V VI V I. Of particular significance are the blocks of 4 chord progressions and the presence of a V I (a perfect cadence) at the end of the sequence. Also note that the chords are only placed on the strong (strength=”1.0”) beats.

```
<!-- ... -->
&lt;segment from="0" to="1378"&gt;
  <landmark at="0">
    <-- ... -->
    <marker strength="1.0" type="beat"/>
  </landmark>
  <landmark at="17">
    <marker strength="0.25" type="beat"/>
  </landmark>
  <landmark at="34">
    <marker strength="1.0" type="beat"/>
  </landmark>
  <landmark at="51">
    <marker strength="0.25" type="beat"/>
  </landmark>
  <landmark at="68">
    <marker strength="1.0" type="beat"/>
  </landmark>
  <landmark at="85">
    <marker strength="0.25" type="beat"/>
  </landmark>
  <landmark at="102">
    <marker strength="0.5" type="beat"/>
  </landmark>
  <landmark at="119">
    <marker strength="0.25" type="beat"/>
  </landmark>
  <landmark at="136">
    <marker strength="1.0" type="beat"/>
  </landmark>
  <!-- ... -->
&lt;/segment&gt;
```

Listing 9.14: The pulse landmarks generated for the first shot of the scene.

```
<!-- ... -->
  <part id="P1">
    <measure number="1">
      <attributes>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
      </attributes>
      <direction placement="above">
        <direction-type>
          <words font-weight="bold" relative-x="-40">Andante</words>
        </direction-type>
        <sound tempo="88"/>
      </direction>
    </measure>
    <measure number="2">
    </measure>
    <measure number="3">
    </measure>
    <!-- ... -->
  </part>
</score-partwise>
```

Listing 9.15: The MusicXML time signature and measures generated for the first shot of the scene.

```
<!-- ... -->
&lt;segment from="0" to="1378"&gt;
  <landmark at="0">
    <!-- ... -->
    <marker strength="1.0" type="beat"/>
    <marker ref="Dm" type="key" />
  </landmark>
  <landmark at="17">
    <marker strength="0.25" type="beat"/>
    <!-- ... -->
&lt;/segment&gt;
```

Listing 9.16: The key landmark generated for the first shot of the scene.

```
<!-- ... -->
    <measure number="1">
      <attributes>
        <key>
          <fifths>-1</fifths>
          <mode>minor</mode>
        </key>
        <time>
          <beats>4</beats>
    <!-- ... -->
  </part>
</score-partwise>
```

Listing 9.17: The MusicXML key signature generated for the first shot of the scene.

```
<!-- ... -->
&lt;segment from="0" to="1378"&gt;
  <landmark at="0">
    <-- ... -->
    <marker ref="Dm" type="key"/>
    <marker ref="II" type="chord" />
  </landmark>
  <landmark at="17">
    <marker strength="0.25" type="beat"/>
  </landmark>
  <landmark at="34">
    <marker strength="1.0" type="beat"/>
    <marker ref="V" type="chord" />
  </landmark>
  <landmark at="51">
    <marker strength="0.25" type="beat"/>
  </landmark>
  <landmark at="68">
    <marker strength="1.0" type="beat"/>
    <marker ref="II" type="chord" />
  </landmark>
  <!-- ... -->
&lt;/segment&gt;
```

Listing 9.18: The chord landmarks placed on the strong beats of the segment.

9.7.5 Instrumentation Agent

The instrumentation agent is the point at which the composition splits into a more complex process. The previous stages are applicable to all instruments, whereas the successive stages must be repeated for each part. The parts selected for this piece are notably those chosen in the landmark stage, ordered by probability. Listing 9.19 shows the landmarks inserted by the instrumentation agent - namely parts for strings, woodwind, and brass, while Listing 9.20 shows the resultant MusicXML version.

```
<!-- ... -->
&lt;segment from="0" to="1378"&gt;
  <landmark at="0">
    <-- ... -->
    <marker ref="Dm" type="key"/>
    <marker ref="II" type="chord" />
    <marker ref="strings" type="instrumentation"/>
    <marker ref="woodwind" type="instrumentation"/>
    <marker ref="brass" type="instrumentation"/>
  </landmark>
  <!-- ... -->
&lt;/segment&gt;
```

Listing 9.19: The instrument landmarks placed at the start of the segment.

9.7.6 Rhythm Agent

The first agent to work on a by-part basis, the rhythm agent inserts many more markers into the landmark file (see Listing 9.21). These note markers include the durations, and new landmarks are created where necessary. Note that the dotted quaver and double-quaver rhythms are both used here, with the brass part providing a double-quaver underlay. Had single notes been included in the landmark file, some extra variation could have been produced. Later parts of the output also include the crotchet rhythm, while other rhythms continue in parallel.

9.7.7 Melody Agent

The final agent uses the information from the key, chord, beat, and rhythm agent to fill in pitch information for the individual notes (see Figure 9.3 and Listings 9.23 and 9.24). While there are a few clashes (the F and G in the second landmark, for example), the E G Bb chord (chord II) can be seen to have influenced the selection of notes. Furthermore, the fifths suggested in the modifiers for the brass section are visible in the A D progression over the third and fourth landmark.

```
<score-partwise>
  <movement-title>Untitled</movement-title>
  <part-list>
    <score-part id="P1">
      <part-name>Strings</part-name>
    </score-part>
    <score-part id="P2">
      <part-name>Woodwind</part-name>
    </score-part>
    <score-part id="P3">
      <part-name>Brass</part-name>
    </score-part>
  </part-list>

  <part id="P1">
    <measure number="1">
      <attributes>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
      </attributes>
      <direction placement="above">
        <direction-type>
          <words font-weight="bold" relative-x="-40">Andante</words>
        </direction-type>
        <sound tempo="88"/>
      </direction>
    </measure>
    <measure number="2">
    <!-- ... -->
  </part>
  <part id="P2">
    <!-- ... -->
  </part>
  <part id="P3">
    <!-- ... -->
  </part>
</score-partwise>
```

Listing 9.20: The MusicXML with additional instrument parts.

9.8 Evaluation

While the musical agents within SBS are functional, this work has focused on the design and implementation of the infrastructure of SBS, and hence full evaluation has not been undertaken. There are effectively two stages to the evaluation of the SBS output: whether the music produced is listenable, and whether the music is suitable for the media it is accompanying. Due to the design of the system, it would also be possible to feed this evaluation back into the evolution process for subsequent generations.

For the first stage, a technique proposed by Unehara[5] could be applied. This suggests a subjective music evaluation process with three classes of evaluation: Total evaluation, where users listen to presented musical works and evaluate the whole based on their feeling (as ‘good’, ‘very good’, ‘neutral’ or ‘bad’); Partial evaluation, where areas of the music are selected and rated; and finally a choice of what the listener believes is the best work in the collection, which is put forward into the next population.

[5] Unehara and Onisawa [2004]

```
<!-- ... -->
    <marker instrument="strings" length="0.75" type="note" />
    <marker instrument="woodwind" length="0.75" type="note" />
    <marker instrument="brass" length="0.75" type="note" />
  </landmark>

  <landmark at="13">
    <marker instrument="strings" length="0.25" type="note" />
    <marker instrument="woodwind" length="0.25" type="note" />
    <marker instrument="brass" length="0.25" type="note" />
  </landmark>

  <landmark at="17">
    <marker strength="0.25" type="beat"/>
    <marker instrument="strings" length="0.75" type="note" />
    <marker instrument="woodwind" length="0.75" type="note" />
    <marker instrument="brass" length="0.5" type="note" />
  </landmark>

  <landmark at="26">
    <marker instrument="brass" length="0.5" type="note" />
  </landmark>

  <landmark at="30">
    <marker instrument="strings" length="0.25" type="note" />
    <marker instrument="woodwind" length="0.25" type="note" />
  </landmark>
  <!-- ... -->
&lt;/segment&gt;
```

LISTING 9.21: The instrument landmarks placed at the start of the segment.

The evaluation of the suitability of the music requires a different testing scenario. Ideally the film should be viewed together with the composed music, and feedback attained from the viewers after the viewing. Alternatively, the viewer could be provided with a means to give instant feedback during the movie to gauge the response to specific moments in the score.

9.9 Summary

As is evident from this case study, SBS covers a large area of the composition process from the media annotation to the individual musical generation stages. Naturally, there are areas that may be improved, but the framework is flexible enough to allow for this. At present the annotation is manual, although the use of automatic techniques to locate shot transitions eased this process considerably, taking only a few hours to annotate the shot used in the example. The SiX markup was also very straightforward, using an existing portion of the screenplay and simply adding tags where appropriate.

Following initial markup, the composer representation was created. This was a slower task, but one which would be aided with a set of presets and a tool for the easy creation and preview of modifiers. Furthermore, the mapping of queries to modifiers could be eased with an interface to hide the RDQL, instead providing a querying tool designed for the OntoMedia ontologies.

```
<!-- ... -->
  <part id="P1">
    <measure number="1">
      <!-- ... -->
      <note>
        <duration>0.75</duration>
      </note>
      <note>
        <duration>0.25</duration>
      </note>
      <note>
        <duration>0.75</duration>
      </note>
      <note>
        <duration>0.25</duration>
      </note>
      <note>
        <duration>0.75</duration>
      </note>
      <note>
        <duration>0.25</duration>
      </note>
      <note>
        <duration>1.0</duration>
      </note>
      <!-- ... -->
  </part>
  <part id="P2">
    <measure number="1">
      <!-- ... -->
      <note>
        <duration>0.75</duration>
      </note>
      <note>
        <duration>0.25</duration>
      </note>
      <note>
        <duration>0.75</duration>
      </note>
      <note>
        <duration>0.25</duration>
      </note>
      <note>
        <duration>0.5</duration>
      </note>
      <note>
        <duration>0.5</duration>
      </note>
      <note>
        <duration>0.5</duration>
      </note>
      <note>
        <duration>0.5</duration>
      </note>
      <!-- ... -->
  </part>
  <part id="P3">
    <!-- as above -->
  </part>
</score-partwise>
```

Listing 9.22: The MusicXML with note durations.

FIGURE 9.3: The melody generated for the first segment.

```
<!-- ... -->
    <marker instrument="strings" length="0.75" type="note" pitch="Bb" octave="4"
     />
    <marker instrument="woodwind" length="0.75" type="note" pitch="G" octave="4"
     />
    <marker instrument="brass" length="0.75" type="note" pitch="E" octave="4" />
  </landmark>

  <landmark at="13">
    <marker instrument="strings" length="0.25" type="note" pitch="F" octave="4"/>
    <marker instrument="woodwind" length="0.25" type="note" pitch="Bb" octave="4"
    />
    <marker instrument="brass" length="0.25" type="note" pitch="G" octave="4"/>
  </landmark>

  <landmark at="17">
    <marker strength="0.25" type="beat"/>
    <marker instrument="strings" length="0.75" type="note" pitch="D" octave="4"/>
    <marker instrument="woodwind" length="0.75" type="note" pitch="C" octave="4"
    />
    <marker instrument="brass" length="0.5" type="note" pitch="A" octave="4"/>
  </landmark>

  <landmark at="26">
    <marker instrument="brass" length="0.5" type="note" pitch="D" octave="4"/>
  </landmark>

  <landmark at="30">
    <marker instrument="strings" length="0.25" type="note" pitch="G" octave="4"/>
    <marker instrument="woodwind" length="0.25" type="note" pitch="G" octave="4"
    />
  </landmark>
  <!-- ... -->
&lt;/segment&gt;
```

LISTING 9.23: The first set of notes generated by the melody agent.

```
<!-- ... -->
  <part id="P1">
    <measure number="1">
      <!-- ... -->
      <note>
        <duration>0.75</duration>
        <step>B</step>
        <alter>-1</alter>
        <octave>4</octave>
      </note>
      <note>
        <duration>0.25</duration>
        <step>F</step>
        <octave>4</octave>
      </note>
      <note>
        <duration>0.75</duration>
        <step>D</step>
        <octave>4</octave>
      </note>
      <note>
        <duration>0.25</duration>
        <step>G</step>
        <octave>4</octave>
      </note>
      <!-- ... -->
  </part>
```

Listing 9.24: The initial notes for the string part in MusicXML format.

Once the various configuration files were complete, the agent graph was employed. This was a simple process, with just the changing of a filename required to inform the agent graph as to the location of the landmark file. On running, the agent graph triggered the various composing agents where necessary. This was expensive on a CPU usage/agent basis, but the separate agents could be easily distributed over a cluster to reduce this. Time-wise, each agent only took a few minutes to complete its task, although this was on a comparatively short segment so distribution would again be beneficial. Also, the agents allow for configuration of generation numbers, with higher numbers providing results closer to the requirements but also taking more time.

At present the rhythm and melody operators have the most room for improvement. As was mentioned previously, allowing for rhythm triggers in the landmark file would improve the selection process, and using a transition approach for the segment changing would provide more flexibility for the composer. For the melody agent, extra operators to improve the generated melody would be beneficial, such as repetition operators to ensure that melody fragments recur. Secondly, an extra parameter could be available to suggest the complexity of the layers, with a low value shifting the music towards homophonic (chordal) music and a high value tending towards polyphony. Finally, the addition of a motive agent (as described in the next chapter) would provide themes that recur throughout the piece.

Chapter 10

Conclusions

10.1 Overall Conclusions

The State-Based Sequencer is an exceptionally broad project, spanning from the annotation of film to the automatic composition of music. As such, it has warranted the creation of a number of specialist tools, and the introduction of many novel concepts.

OntoMedia, which is used at the initial annotation stage, is a powerful ontology. It provides the ability to not only annotate film, but also any other temporal forms. Whether chemical interactions, mythology, or narrative, its hierarchical approach allows for its annotation. Different media sources may be compared at the lowest level, thus enabling the analysis of similar elements of otherwise incompatible items. Several further technologies have been created on top of the OntoMedia ontology, including Meditate for character annotation and the OntoMedia Entity Store for the collaborative creation of annotations. Parallel to OntoMedia, the SiX schema was designed, allowing for the simple description of screenplays, a format which through OntoMedia references can be leveraged for script querying.

Central to SBS is the Light Agent Framework. This was created to be portable and efficient, while providing a fast development time for powerful agents. As well as being used as the foundation for the musical agents in the State-Based Sequencer, the C++ implementation of LAF has been employed as part of a project for multi-camera computer vision. A Python version has also been created, which, with Java, gives three interoperable versions of the framework. Several of the features, including the agent graph functionality and the ability to autolocate routers via zeroconf, were beneficial to the development of the sequencer.

Finally, the compositional techniques created for SBS are a significant step in the field of algorithmic composition. Utilising a distributed set of composing agents allows for the task of musical generation to be decomposed into elements that are closely analogous

to that of a traditional composer. Furthermore, by separating the agents in this way, it is simple to adapt the composing algorithms used by the different stages. If a new approach to melody composition is proposed, it can simply be swapped in to the agent network. The landmark file format and composer mappings, also designed as part of this project, allow for an easily-parsed, portable format for communications between these agents, hence further easing the addition of new techniques.

10.2 Future Work

While SBS is able to perform adequately at the composition process, there is still future work that may be carried out. As was discussed previously, the rhythm and melody agents could be fine tuned to provide better results, with the former possibly using a graph-based segment selector with rhythm triggers and the latter using more advanced operators for melody generation with a fitness function that takes into account the melodies of each instrument. The creation of a motif agent would also be beneficial, ensuring that themes for concepts within the media are preserved throughout the soundtrack.

As the Light Agent Framework, OntoMedia, and the SBS XML formats have been implemented, it is now possible to carry out more composition tests and evaluate the existing agents. The SBS project has been primarily focused on the development of these component technologies, with an aim to provide the means to develop composition agents to incorporate new techniques into the system, so it is straightforward to compare the music produced using different algorithm types.

The annotation process would benefit from further automation. This would ideally occur during the creation process, with tools to annotate the film during production and the script during writing. The additional context available at these stages would ease the task - for example, shooting and location information could provide useful parameters to algorithms for actor and environment recognition. Efforts are ongoing in this area, especially with the growth of the semantic web and the need for annotated media.

The only performance limits of SBS at present are the size of the landmark file and the number of evolutions carried out in the agents. The former may be handled by running the framework over a cluster, with individual agents handling different sections of the score - a task for which the agent framework is designed. The latter would require either a more powerful host, or a means to distribute the GA over a number of machines. There are, however, no fundamental problems with the system, and the flexibility of the approach allows for simple upgrade should better approaches to composition be found.

One potential future use of SBS would be that of a composer guidance system. Rather than producing a final piece of music, this would create additional annotation for the

media with suggestions as to the direction the music could take. Making use of the composer representation it might suggest a certain key for a passage, or a rhythmic idea. This annotation could be imported into composing software to provide notes in the score, or combined with the SiX screenplay representation to associate it more closely with the script.

While the agent framework is designed for non-real-time composition, it would be possible to use the components of SBS for real-time applications such as computer games. OntoMedia is, by design, capable of annotating plot lines, and the composer representation can join with this to provide musical direction for the current event within the media. The GA approach to composing would likely be too inefficient for this task, however, so a system using preprepared segments which may be modified could be more suitable.

From OntoMedia and SiX for script annotation, to the Light Agent Framework for agent development, and to the distributed musical agents specialising in composing tasks, the State-Based Sequencer is a unique and powerful approach to musical composition. In addition, the projects created as part of the system are now in use in other applications, providing a valuable contribution to future research.

10.3 Top-Down Composition

The music currently obtained from SBS is playable and listenable, but is often unadventurous (i.e. conforms too closely to the parameters) or too random. One possible approach to rectifying this would be a top-down approach to composition.

At present, the compositional model used by SBS commences with tempo, thence moving through timing information, key information, and then into more complex agents for rhythm and melody. This is effectively a bottom-up approach, beginning with a blank score, filling in bass information (i.e. the chord agent) and only then considering rhythm and melody. A top-down approach would begin with melody, creating a set of key-independent segments. These could be per-concept (i.e. motifs) or representing a combination of concepts (such as a character in a certain location). The rhythm agent would follow the melody agent, selecting note lengths for the pitches.

After the rhythmic assignment the system would then move onto the chord agent. This could be a variation of the existing chord agent, with the engine matching the relevant chords to the notes of the melody rather than to the key information. Finally, the key agent would select appropriate points for key changes based on the selected chord progressions, and the key-independent melodic segments would be converted into notes.

References

M. Bal. *Introduction to the theory of Narrative*. University of Toronto Press, second edition, 1997.

J. Beran. Music - chaos, fractals, and information. *Chance*, 17(4), 2004.

J. Biles. Genjam: A genetic algorithm for generating jazz solos, 1994. URL citeseer.nj.nec.com/biles94genjam.html.

W. Buxton, W. Reeves, R. Baeker, and L. Mezei. The use of hierarchy and instance in a data structure for computer music. *Computer Music Journal*, 2(2):10–20, 1978.

S. Chatman. *Story and Discourse*. Cornell University Press, New York, 1978.

Y. Chen and E. K. Wong. Augmented image histogram for image and video similarity search. In *Proc. SPIE Conf. Storage and Retrieval for Image and Video Database VII*, pages 523–532, 1999.

N. Crofts, M. Doerr, T. Gill, S. Stead, and M. S. (eds). Definition of the CIDOC CRM conceptual reference model. Reference document, International Council of Museums, March 2005. URL http://cidoc.ics.forth.gr/official_release_cidoc.html.

P. K. Dick. *The Preserving Machine*, chapter We Can Remember It for You Wholesale. Ace Books, 1969.

W. K. Everson. *American Silent Film*. Oxford University Press, 1978.

R. Ewerth, M. Schwalb, P. Tessmann, and B. Freisleben. Estimation of arbitrary camera motion in mpeg videos. In *Pattern Recognition*, pages 512–515, August 2004.

M. Farbood and B. Schoner. Analysis and synthesis of Palestrina-style counterpoint using Markov chains. In *International Computer Music Conference*, 2001. URL citeseer.nj.nec.com/mishra95mapping.html.

A. Fitzgibbon and A. Zisserman. On affine invariant clustering and automatic cast listing in movies. In *ECCV*, pages 304–320, 2002. URL citeseer.nj.nec.com/fitzgibbon02affine.html.

M. A. Framingham. *The Common Object Request Broker: Architecture and Specification, Second Edition*. 1999.

M. Good. MusicXML: An internet-friendly format for sheet music. In *XML Conference and Expo*, 2001. URL http://www.idealliance.org/papers/xml2001/papers/html/03-04-05.html.

M. J. Grant. *Serial Music, Serial Aesthetics: Compositional Theory in Post-War Europe*. Cambridge University Press, 2001.

J. H. Holland. *Adaptation in Natural and Artificial Systems*. Ann Arbor: University of Michigan Press, 1975.

A. Horner and D. E. Goldberg. Genetic algorithms and computer-assisted music composition. In *ICGA*, 1991.

D. Horowitz. Generating rhythms with genetic algorithms. In *Proceedings of International Computer Music Conference (ICMC94)*, pages 142–143, 1994.

J. Hunter. Enhancing the semantic interoperability of multimedia through a core ontology. *IEEE Transactions on Circuits and Systems for Video Technology*, January 2003. URL http://archive.dstc.edu.au/RDU/staff/jane-hunter/events/paper.html.

B. Jacob. Composing with genetic algorithms, 1995. URL citeseer.nj.nec.com/jacob95composing.html.

M. O. Jewell, K. F. Lawrence, M. M. Tuffield, A. Prügel-Bennett, D. E. Millard, M. S. Nixon, m c schraefel, and N. R. Shadbolt. Ontomedia: An ontology for the representation of heterogeneous media. In *Multimedia Information Retrieval Workshop (MMIR 2005) SIGIR*, 2005a.

M. O. Jewell, L. Middleton, M. S. Nixon, A. Prügel-Bennett, and S. C. Wong. A distributed approach to musical composition. *9th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES'05) Part III*, 3683:642–648, 2005b.

M. O. Jewell, M. S. Nixon, and A. Prügel-Bennett. CBS: A concept-based sequencer for soundtrack composition. In *WEDELMUSIC*, 2003.

M. O. Jewell, M. S. Nixon, and A. Prügel-Bennett. State-based sequencing: Directing the evolution of music. In *International Computer Music Conference*, 2005c.

K. Jones. Compositional applications of stochastic processes. *Computer Music Journal*, 5(2):45–61, 1981.

K. Kalinak. *Settling the Score: Music and the Classical Hollywood Film*. University of Wisconsin Press, 1992.

F. Karlin. *On the Track: A Guide to Contemporary Film Scoring*. Routledge, 2004.

H. Kirchmeyer. On the historical construction of rationalistic music. *Die Reihe*, 8:11–29, 1962.

K. F. Lawrence, M. O. Jewell, m c schraefel, and A. Prügel-Bennett. Annotation of heterogenous media using ontomedia. In *First International Workshop on Semantic Web Annotations for Multimedia (SWAMM)*, 2006.

K. F. Lawrence, M. M. Tuffield, M. O. Jewell, A. Prügel-Bennett, D. E. Millard, M. S. Nixon, m c schraefel, and N. R. Shadbolt. Ontomedia - creating an ontology for marking up the contents of heterogeneous media. In *Ontology Patterns for the Semantic Web ISWC-05 Workshop*, 2005.

J. Leach and J. Fitch. Nature, music, and algorithmic composition. *Computer Music Journal*, 19(2):23–33, 1995.

D. G. Loy. *Composing with computers–A survey of some compositional formalisms and music programming languages*, pages 291–396. Current Directions in Computer Music Research. MIT Press, Cambridge, MA, 1989.

B. Maess, S. Koelsch, T. C. Gunter, and A. D. Friederici. Musical syntax is processed in broca's area: an MEG study. *Nature Neuroscience*, 4(5):540–545, 2001.

J. McCormack. Grammar based music composition. In *Complex Systems*, 1996.

L. Middleton, S. C. Wong, M. O. Jewell, J. N. Carter, and M. S. Nixon. Lightweight agent framework for camera array applications. *9th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES'05) Part IV, Lecture Notes in Computer Science*, 3684:150–156, 2005a.

L. Middleton, S. C. Wong, M. O. Jewell, J. N. Carter, and M. S. Nixon. A middleware for a large array of cameras. In *EEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3682–3687, 2005b.

I. C. Millard, D. C. DeRoure, and N. R. Shadbolt. The use of ontologies in contextually aware environments. In *First International Workshop on Advanced Context Modelling, Reasoning and Management*, 2004.

K. Minami, A. Akutsu, H. Hamada, and Y. Tonomura. Video handling with music and speech detection. *IEEE MultiMedia*, 5(3):17–25, 1998. ISSN 1070-986X.

E. R. Miranda. *Composing Music with Computers*. Oxford: Focal Press, 2001.

E. R. Miranda. On the music of emergent behavior: What can evolutionary computation bring to the musician? *Leonardo*, 36(1):55–59, 2003.

M. Rabiger. *Directing: Film Techniques and Aesthetics*. Focal Press, 1997.

C. Roads. Grammars as representations for music. *Computer Music Journal*, 3(1):45–55, 1979.

C. Roads. *Computer Music Tutorial.* MIT Press, 2001.

J. Robertson, T. Stapleford, A. Quincey, and G. Wiggins. Real-time music generation for a virtual environment. In *ECAI 98 Workshop on AI/ALife and Entertainment*, 1998.

M. Russ. *Mussorgsky: Pictures at an Exhibition.* Cambridge University Press, 1992.

M. V. Srinivasan, S. Venkatesh, and R. Hosie. Qualitative estimation of camera motion parameters from video sequences. *Pattern Recognition*, 30(4):593–606, 1997.

K. Thywissen. Genotator: An environment for investigating the application of genetic algorithms in computer assisted composition. In *Proceedings of International Computer Music Conference (ICMC96)*, pages 274–277, 1996.

M. Unehara and T. Onisawa. Music composition system with human evaluation as human centered system. *Soft Computing*, 7(3):167–178, 2004.

H. von Koch. Sur une courbe continue sans tangente, obtenue par une construction géométrique élémentaire. *Archiv för Matemat., Astron. och Fys*, 1:681–702, 1904.

R. F. Voss and J. Clarke. 1/f noise in music and speech. *Nature*, 258:23–33, 1975.

Y. Wang, Z. Liu, and J.-C. Huang. Multimedia content analysis. *IEEE Signal Processing Magazine*, pages 12–36, 2000.

G. Wiggins, G. Papadopoulos, S. Phon-Amnuaisuk, and A. Tuson. Evolutionary methods for musical composition. *International Journal of Computing Anticipatory Systems*, 1999.

E. Wold, T. Blum, D. Keislar, and J. Wheaten. Content-based classification, search, and retrieval of audio. *IEEE Multimedia*, 3(3):27–36, 1996.

S. Wolfram. Universality and complexity in cellular automata. *Physica D*, 10:1–35, 1984.

S. Yuan-Yuan, W. Xue, and S. Bin. Several features for discrimination between vocal sounds and other environmental sounds. In *Proceedings of the European Signal Processing Conference*, 2004.

R. Zabih, J. Miller, and K. Mai. A feature-based algorithm for detecting and classifying production effects. *Multimedia Systems*, 7:119–128, 1999.

Appendix A

The OntoMedia Core Ontology

```
<owl:Class rdf:ID="Expression">
  <rdfs:label>Expression</rdfs:label>
  <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
  This class represents a piece of information conveyed through
  a medium
  </rdfs:comment>
</owl:Class>

<owl:ObjectProperty rdf:ID="inspired-by">
  <rdfs:label>inspired by</rdfs:label>
  <rdfs:comment rdf:datatype="&xsd;string">
  This property indicates that the expression
  was inspired by another
  </rdfs:comment>
  <rdfs:domain rdf:resource="#Expression"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="inspired"/>
  </owl:inverseOf>
  <rdfs:range rdf:resource="#Expression"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="has-variant">
  <rdfs:label>has variant</rdfs:label>
  <rdfs:comment rdf:datatype="&xsd;string">
  This property indicates that the expression
  is a variation of another
  </rdfs:comment>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="is-variant-of"/>
  </owl:inverseOf>
  <rdfs:range rdf:resource="#Expression"/>
  <rdfs:domain rdf:resource="#Expression"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="is-potentially">
  <rdfs:label>is potentially</rdfs:label>
  <rdfs:range rdf:resource="#Expression"/>
  <rdfs:domain rdf:resource="#Expression"/>
  <rdfs:comment rdf:datatype="&xsd;string">
  This property indicates that the expression is
```

```
    potentially another. For example, it may be a
    possible future version
    </rdfs:comment>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="is">
    <rdfs:label>is</rdfs:label>
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="is-not">
        <rdfs:comment rdf:datatype="&xsd;string">
  This property indicates that the expression
  is entirely different to another
  </rdfs:comment>
      </owl:ObjectProperty>
    </owl:inverseOf>
    <rdfs:domain rdf:resource="#Expression"/>
    <rdfs:range rdf:resource="#Expression"/>
    <rdfs:comment rdf:datatype="&xsd;string">
    This property indicates that the expression
    is exactly the same as another
    </rdfs:comment>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="in-context">
    <rdfs:label>in context</rdfs:label>
    <rdfs:comment rdf:datatype="&xsd;string">
    This property specifies the context in which
    this expression lies.
    </rdfs:comment>
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="includes-expression"/>
    </owl:inverseOf>
    <rdfs:range rdf:resource="#Context"/>
    <rdfs:domain rdf:resource="#Expression"/>
 </owl:ObjectProperty>

  <!-- Items -->

  <owl:Class rdf:ID="Item">
    <rdfs:comment rdf:datatype="&xsd;string">This class represents an
    entity which may participate in an event within the media. An Item may
    be abstract or physical</rdfs:comment> <rdfs:label>Item</rdfs:label>
    <rdfs:subClassOf rdf:resource="#Entity" />
  </owl:Class>

  <owl:Class rdf:ID="Physical-Item">
    <rdfs:comment rdf:datatype="&xsd;string">This class represents
    a physical entity which may participate in an event within the
    media</rdfs:comment> <rdfs:label>Physical Item</rdfs:label>
    <rdfs:subClassOf rdf:resource="#Item" />
  </owl:Class>

  <owl:Class rdf:ID="Abstract-Item">
    <rdfs:comment rdf:datatype="&xsd;string">This class represents
    an abstract entity which may participate in an event within
    the media</rdfs:comment> <rdfs:label>Abstract Item</rdfs:label>
    <rdfs:subClassOf rdf:resource="#Item" />
  </owl:Class>
```

```
<owl:Class rdf:ID="Context">
  <rdfs:comment rdf:datatype="&xsd;string">This class represents
  the context in which an event or entity exists</rdfs:comment>
  <rdfs:label>Context</rdfs:label> <rdfs:subClassOf
  rdf:resource="#Abstract-Item" />
</owl:Class>

<owl:Class rdf:ID="Collection">
  <rdfs:comment rdf:datatype="&xsd;string">This class
  represents a collection of entities</rdfs:comment>
  <rdfs:label>Collection</rdfs:label> <rdfs:subClassOf
  rdf:resource="#Abstract-Item" />
</owl:Class>

<!-- Temporal -->

<owl:Class rdf:ID="Timeline">
  <rdfs:comment rdf:datatype="&xsd;string">This class
  contains a sequence of occurring events</rdfs:comment>
  <rdfs:label>Timeline</rdfs:label> <rdfs:subClassOf
  rdf:resource="#Entity" />
</owl:Class>

<owl:Class rdf:ID="Occurrence">
  <rdfs:comment rdf:datatype="&xsd;string">This class represents a single
  occurrence of an event, placing it at a position in a timeline</rdfs:comment>
  <rdfs:label>Occurrence</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Entity" />
</owl:Class>

<owl:ObjectProperty rdf:ID="location">
  <rdfs:comment rdf:datatype="&xsd;string">This location in which this event
  occurs.</rdfs:comment>
  <rdfs:range rdf:resource="&oms;#Space"/>
  <rdfs:domain rdf:resource="#Event" />
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="precedes">
  <rdfs:label>precedes</rdfs:label>
  <rdfs:comment rdf:datatype="&xsd;string">This property defines the occurrence
   which immediately follows this occurrence</rdfs:comment>
  <rdfs:range rdf:resource="#Occurence"/>
  <rdfs:domain rdf:resource="#Occurence"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="follows"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="follows">
  <rdfs:label>follows</rdfs:label>
  <rdfs:range rdf:resource="#Occurence"/>
  <rdfs:domain rdf:resource="#Occurence"/>
  <owl:inverseOf rdf:resource="#precedes"/>
  <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">Follows
  should specify both timeline and event IDs where there is more than one
  timeline or over two events</rdfs:comment>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="timeline-ref">
```

```
  <rdfs:range rdf:resource="#Timeline"/>
  <rdfs:domain rdf:resource="#Occurence"/>
</owl:ObjectProperty>

<!-- Events -->

<owl:Class rdf:ID="Event">
  <rdfs:label>Event</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Expression" />
</owl:Class>

<owl:ObjectProperty rdf:ID="has-subject-entity">
  <rdfs:label>has subject entity</rdfs:label>
  <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">This
  property specifies the entity which carries out the aim of the event</
  rdfs:comment>
  <rdfs:domain rdf:resource="#Event"/>
  <rdfs:range rdf:resource="#Entity"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="has-object-entity">
  <rdfs:label>has object entity</rdfs:label>
  <rdfs:comment rdf:datatype="http://www.w3.org/2001/XMLSchema#string">This
  property specifies the entity which is the target of the event</rdfs:comment>
  <rdfs:range rdf:resource="#Entity"/>
  <rdfs:domain rdf:resource="#Event"/>
</owl:ObjectProperty>


<owl:ObjectProperty rdf:ID="has-occurrence">
  <rdfs:label>has occurrence</rdfs:label>
  <rdfs:comment rdf:datatype="&xsd;string">This property defines any
  occurrences of this event</rdfs:comment>
  <rdfs:range rdf:resource="#Occurrence"/>
  <rdfs:domain rdf:resource="#Event" />
  <owl:inverseOf>
    <owl:FunctionalProperty rdf:ID="occurrence-of"/>
  </owl:inverseOf>
</owl:ObjectProperty>


<owl:DatatypeProperty rdf:ID="summary">
  <rdfs:label>summary</rdfs:label>
  <rdfs:domain rdf:resource="#Event"/>
  <rdfs:range rdf:resource="&xsd;string"/>
  <rdfs:comment rdf:datatype="&xsd;string">This property is a plain-text
  description of what occurs in the event</rdfs:comment>
</owl:DatatypeProperty>

<owl:ObjectProperty rdf:ID="precondition">
  <rdfs:label>precondition</rdfs:label>
  <rdfs:range>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Entity"/>
        <owl:Class rdf:about="#Event"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:range>
```

```
    <rdfs:domain rdf:resource="#Event"/>
    <rdfs:comment rdf:datatype="&xsd;string">This property is a state that must
    exist before the event can occur</rdfs:comment>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="postcondition">
    <rdfs:label>postcondition</rdfs:label>
    <rdfs:range>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#Entity"/>
          <owl:Class rdf:about="#Event"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:range>
    <rdfs:comment rdf:datatype="&xsd;string">This property contains the state
    which should occur as a consequence of this event</rdfs:comment>
    <rdfs:domain rdf:resource="#Event"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="involves">
    <rdfs:label>involves</rdfs:label>
    <rdfs:comment rdf:datatype="&xsd;string">This property specifies the entities
     involved in this event. Note that this includes the subject and object.</
    rdfs:comment>
    <rdfs:range rdf:resource="#Entity"/>
    <rdfs:domain rdf:resource="#Event"/>
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="involved-in" />
    </owl:inverseOf>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="causes">
    <rdfs:label>causes</rdfs:label>
    <rdfs:comment rdf:datatype="&xsd;string">This property indicates the
    instigating factor of an event, whether it be an item, event, or collection.
    </rdfs:comment>
    <rdfs:range>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#Event"/>
          <owl:Class rdf:about="#Entity"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:range>
    <rdfs:domain>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#Event"/>
          <owl:Class rdf:about="#Entity"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="caused_by"/>
    </owl:inverseOf>
  </owl:ObjectProperty>

  <!-- -->
```

```
<owl:Class rdf:ID="Gain">
  <rdfs:label>Gain</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Event" />
  <rdfs:comment rdf:datatype="&xsd;string">This event class results in an
  overall increase of the entities related to the primary subject or subjects
  of the event</rdfs:comment>
</owl:Class>

<owl:Class rdf:ID="Introduction">
  <rdfs:label>Introduction</rdfs:label>
  <rdfs:comment rdf:datatype="&xsd;string">This event class denotes the
  introduction of an entity to the media</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#Event" />
</owl:Class>

<owl:Class rdf:ID="Loss">
  <rdfs:label>Loss</rdfs:label>
  <rdfs:comment rdf:datatype="&xsd;string">This event class results in an
  overall reduction of the entities related to the primary subject or subjects
  of the event</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#Event" />
</owl:Class>

<owl:Class rdf:ID="Transformation">
  <rdfs:comment rdf:datatype="&xsd;string">This event class results in no gain
  or loss of attributes or entities, merely alteration</rdfs:comment>
  <rdfs:label>Transformation</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Event" />
</owl:Class>

<owl:ObjectProperty rdf:ID="from">
  <rdfs:label>from</rdfs:label>
  <rdfs:comment rdf:datatype="&xsd;string">This property specifies the entity
  which is being transformed</rdfs:comment>
  <rdfs:range rdf:resource="#Entity"/>
  <rdfs:domain rdf:resource="#Transformation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="to">
  <rdfs:label>to</rdfs:label>
  <rdfs:comment rdf:datatype="&xsd;string">This property specifies the
  resultant entity</rdfs:comment>
  <rdfs:range rdf:resource="#Entity"/>
  <rdfs:domain rdf:resource="#Transformation"/>
</owl:ObjectProperty>


<owl:Class rdf:ID="Action">
  <rdfs:comment rdf:datatype="&xsd;string">This event class describes an action
   sequence (ie no plot)</rdfs:comment>
  <rdfs:label>Action</rdfs:label>
  <rdfs:subClassOf rdf:resource="#Event" />
</owl:Class>
```