A Distributed Approach to Musical Composition

Michael O. Jewell and Lee Middleton and Mark S. Nixon and Adam Prügel-Bennett and Sylvia C. Wong

University of Southampton, Southampton, SO17 1BJ, UK
moj@ecs.soton.ac.uk

Abstract. Current techniques for automated composition use a single algorithm, focusing on one aspect of musical generation. In our system we make use of several algorithms, distributed using an agent oriented middleware, with each specialising on a separate aspect of composition. This paper describes the architecture and algorithms behind this system, with a focus on the agent framework used for implementation. We show early results which encourage a future application of this framework in automated music composition and analysis.

1 Introduction

Traditionally, algorithmic music composition has aimed to create independent pieces of music using rule-based techniques. These techniques do not have to be handled by a computer - they have been used as far back as 1026 when Guido d'Arezzo assigned pitches to vowel sounds[1] and, more recently, when Ron Pellegrino created music using light hitting wall-mounted photoresistors. However, as computers are now sufficiently powerful, algorithms are often carried out in software.

The burst of computational composing algorithms began with Arnold Schönberg at the start of the 20th century, with Webern and his successors forming serialism from these roots. Iannis Xenakis was a pioneer who, from his 'succès du scandale' *Metastaseis* in 1955, produced multimedia creations based on probability, sonic phenomena, texture, and random generation, and this work contributed to the stochastic approach of composition[2]. Further approaches, such as Voss and Clarke's fractal techniques[3], McAlpine's cellular automata method[1], and Burton's genetic algorithm systems[4] followed, and these make up a collection of 'stock' composition methodologies.

Until recently, however, music composition has concentrated on using single algorithms to control individual elements of the generation process. For example, an algorithm is used to create chord patterns, while the melody, rhythm, and keys are set a priori. Our new distributed approach aims to treat the existing algorithmic techniques as building blocks for the creation of a music composition system, where different algorithms can be plugged in for evaluation.

The distributed composing framework is further bolstered by its strong ties to other media. Rather than generating music with no prior information, a composer model[5] is used to provide a priori information for the algorithms, and a script representation allows for the alteration of these parameters at pertinent points in the bound medium.

This paper is split into two halves. The first provides a technical background to the Light Agent Framework, which is at the core of our system, while the second describes how this framework is used for distributed music composition, with some preliminary results given. Finally, we suggest how we will take advantage of the system for future agents.

2 The Light Agent Framework

The main intention behind our agent framework was simple - that it should be lightweight. To maximise uptake it had to be intuitive for a user to package algorithms, both existing and original, and furthermore it had to be undemanding on the host computer, hence providing the maximum possible resources to the agent.

However, while a streamlined approach was appealing, it was decided to allow for the possibility of adding extra features without disturbing the original interface. This is especially true of the router, which is covered in more detail later in this section. Balancing features and simplicity was key to the development of our system, and we believe it is therefore suitable for a wide variety of applications.

2.1 Agent Design

At the core of every agent in the LAF system is an engine. This was initially designed as a separate module, but was later subsumed into the agent class, partly for ease of threading, and partly to reduce the files required to design an agent. The engine can only be activated by one client at a time (to preserve atomicity) but this is complicated when several parameters are passed at dissimilar times. As such two messages, LOCK and UNLOCK, were created.

When any agent connects to a router, it sends a 'stub' of information. This includes the agent type, the creator name, a description, a version number, and any inputs or outputs that can be accommodated. If a client then wishes to lock an agent, it requests that the router select an unlocked agent, return a unique name, and then prevent other clients from interacting while the agent is locked. Once the client is finished, or if it disconnects unexpectedly, the router can unlock the locked agent to let other clients make use of it. Most importantly, when an agent is locked, only the locking client can alter the input and output parameters. To all other clients, the agent appears immutable.

Ports The modeling of the agent parameters uses a further design feature of the light agent framework - ports. Ports have unique names, with a loose hierarchy provided by '.'s. Five port subcategories are defined:

1. agent.identity.*
   The identity ports are a structured representation of the stub described in the previous section, with name, type, creator, description, and version fields. The name is not set by the user; instead the router generates a unique name from the type when the agent connects. As with port names, type names are hierarchical in nature, with dots delimiting. For example, 'string.concat' is a valid type, as is 'music.composing.genetic'.

2. agent.input.* and agent.output.*
   The input and output ports are responsible for passing data between agents and clients. The input ports are immutable, whereas the output ports can be altered to indicate the results of a process. Agents can be configured to require certain ports, while others can be set as optional. This further allows for the chaining of agents, with execution triggering when all compulsory ports are initialized.
3. agent.state
   The agent.state port is the simplest in the framework, and contains a string representing the state of the agent. This can have one of four values, namely 'waiting', 'ready', 'running', and 'exiting'. The agent enters the waiting state on connection, the ready state when all compulsory ports are set, the running once executed, and the exiting state when disconnected. If monitored for changes, clients can use this to give a high level status indication.
4. agent.call.*
   When running larger tasks, it is useful for clients to be able to monitor the progress of jobs. The call ports facilitate this and provide a lower level alternative to the state ports. Four ports are provided: percent, time.current, time.total, and status. agent.call.percent provides the percent of the job complete as a double value,
   agent.time.current and agent.time.total provide the duration of the current job and the time that the agent has been locked, while agent.call.status is a string describing the state of the agent. The percent and time.* ports are most likely to get updated often, so these are typically polled at set intervals. The status, however, is more suitable for an interrupt approach.

Ports are typed, but these types can be defined by the agent author. The standard base types are provided - Boolean, integer, string, double - and these include validation functions to ensure that no incorrect parameters get passed through to the engine.

Remote Agents To ease the usage of agents by clients, a remote agent interface is provided. This acts as a proxy, and allows for methods to be called on an agent as if the instance was local. All communications are handled via an agent session, connected to the router, and when an agent is locked a remote agent is returned to the client. Only a select few methods are provided, including call (to commence executing the agent), port set and retrieval methods, and functions to request information on the locked agent, such as its port and IP address. Once a remote agent is finished with, it can be unlocked via an agent session method, thus eliminating the need to communicate with the router directly.

2.2 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.

3 Agent-Based Composition

To create our composing system, the process of composition was split into individual tasks. This keeps the system analogous to the traditional approach for music writing. Each of these tasks was then implemented as an agent, allowing for the production of an agent graph connecting them together. Furthermore, this gave the ability to rearrange the system to test different combinations of agents. This section details the standard features in our composition agents, then focuses on the operation of these agents.

3.1 Agent Structure

The agents in our composition system all follow the same model. Each is in the 'music.composing.' hierarchy to distinguish from other agents, and each has 2 inputs and 2 outputs. The landmark port and the MusicXML[6] port are the two inputs to the agents, and modified versions of these arguments are provided on the output.

The landmark file contains sets of meshes and mappings, placed at key points in the source medium. In film, a key point may be where the location changes or a character's personality alters. This information is used to prime the algorithms within the individual agents, with the output port containing modified parameters if necessary (for example, to include the beat information). The file is split into segments, to allow for scenes and shots, and these sections can be defined in frames or seconds.

Where the landmark is used for input, the MusicXML port is the output. MusicXML is used as the music transfer format, as it is both easily parsed and able to contain a high level of detail. Where MIDI represents only 'note on' and 'note off', MusicXML contains structures to describe a wide range of musical attributes, such as note length and articulation. We store the music in a part-wise approach, as each agent can then work on individual parts.

To handle the two inputs, each agent has an initial parsing step. This parses the landmark and MusicXML values into component objects. Once execution has completed there is a final parsing step, where the landmark and MusicXML structures are altered to include the results of the operation, and then they are serialised and the output ports are set. This standardised structure, as shown in Figure 1, simplifies the interconnecting of several agents via the agent graph system, and eases debugging - as only two parsers are required.

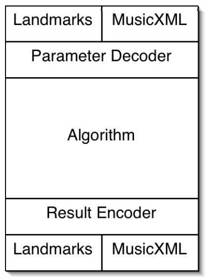

Fig. 1. The standard structure used to represent a musical agent

3.2 Agent Implementation

At present, seven agents have been implemented in the SBS system - namely tempo, pulse, instrumentation, key, chord, rhythm, and melody. Of these seven, six use genetic algorithms to produce the final results, while the other (tempo) only uses a genetic approach when no tempo is provided or when a tempo cannot be easily calculated from event information. These agents, as well as the remaining agents that are under development, are connected to one another in the arrangement shown in Figure 3 using the agent graph system.

The landmark file, described earlier, provides the means for the agent fitness functions to evaluate the population during composition. The musical element is stored in a chromosomal representation; for example, the melody agent stores pitch values within each gene. The tempo, pulse, instrumentation, and rhythm agents use a histogram-based parameter which specifies the probability of each alternative occurring. Taking instrumentation as an example, stringed instruments may have a high probability of occurring, whereas trumpets might have a low probability. In these cases it is trivial for the genetic algorithm to evolve the population in such a way that the most likely combination of options is produced.

The other cases (namely key, chord, and melody) are more complex, as these use a Markov model approach to fitness evaluation. The Markov model specifies the probability of moving from one state to another, with probabilities defined by the composer representation. This state-based approach is essential to these agents, with key changes, chord progressions, and scales being central to the creation of satisfactory music. Again, however, it is not difficult for the genetic algorithm to employ these parameters as constraints in the fitness function. The fitness of the chromosome can be calculated by stepping through the genes, totalling the probabilities of moving from one state to another, with higher results suggesting that the chromosome is more suitable.

The design of the SBS system is such that each stage adds detail to the prior stage: for example, the tempo agent determines the location of beats within the music (and hence the speed), which the pulse agent then supplements with strength information to indicate barline placements. Figure 2 shows the results of this process, with a generated pulse augmented by rhythm and melody.

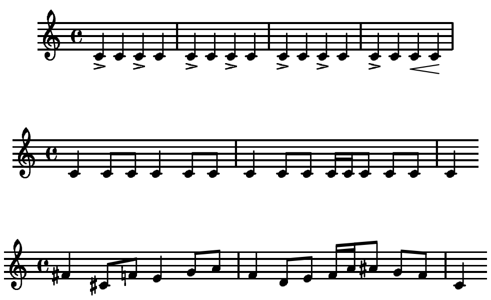

Fig. 2. The creation of a musical phrase using three connected agents. First a pulse is generated, which is then used to generate a suitable rhythm. Finally, a melody is generated and overlaid on the rhythm.

The majority of the stages can also be parallelized to increase efficiency: a film can be split into scenes, and each scene can be handled by a separate collection of agents. There is a need for consistency however, so future work will include sharing musical themes between agents. The Light Agent Framework is ideal for this implementation, as a single router can ensure that messages are passed efficiently between the various agents, as well as selecting available agents to process the landmark representations.

4 Conclusions

The Light Agent Framework is a very elegant approach to a difficult problem - the distribution of applications using multiple components. While typical systems can require extensive configuration, our framework allows for applications to be coded away from the API and then simply dropped in as agents. The framework is also very portable, with versions available in Java, C++, and Python, all using the same user-side function prototypes.

When applied to music composition, LAF is the ideal solution to allow for the task. Though current approaches to music composition are largely sequential the generation of music is by its nature distributed, and our implementation of genetic algorithms encapsulated within individual composing agents proves this fact. Our system moves away from the typical monolithic approach to automated composition, and also allows for the direction of the music by existing temporal media. Furthermore, the framework provides the facilities necessary for the parallelisation of many of the composition tasks, hence reducing processing time.

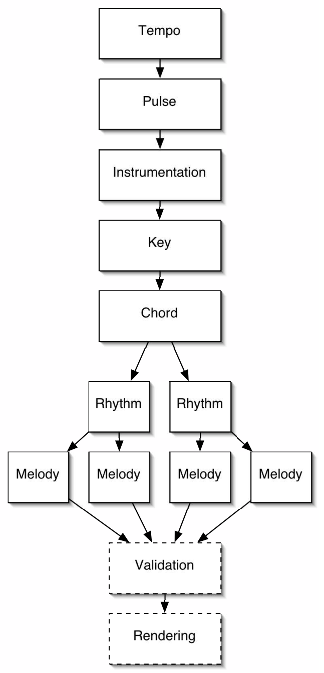

Fig. 3. A structure for the distributed composition of music. Future agents have dashed borders.

Two further agents, one for music validation and one for music rendering, are already at the design stage, so we hope to obtain complete generated scores for further testing. The validation stage is particularly suitable for the agent framework, as we plan to generate a large number of possible scores and use these as the bootstrap for a genetic algorithm. This will require tens, if not hundreds, of melodies, and hence will involve many melody agents communicating with several accumulation agents, with these communicating with the validation agent.

References

1. McAlpine, K., Miranda, E., Hoggar, S.: Making music with algorithms: A case-study system. Computer Music Journal **23** (1999) 19–30
2. Harley, J.: The electroacoustic music of Iannis Xenakis. Computer Music Journal **26** (2004) 33–57
3. Voss, R.F., Clarke, J.: 1/f noise in music and speech. Nature **258** (1975) 23–33
4. Burton, A.R., Vladimirova, T.: Generation of musical sequences with genetic techniques. Computer Music Journal **23** (1999) 59–73
5. Jewell, M.O., Nixon, M.S., Prügel-Bennett, A.: CBS: A concept-based sequencer for soundtrack composition. In: WEDELMUSIC. (2003)
6. Good, M.: MusicXML: An internet-friendly format for sheet music. In: XML Conference and Expo. (2001)