MUSIDO: A Framework for Musical Data Organization to Support Automatic Music Composition

Ryan Demopoulos and Michael Katchabaw
Department of Computer Science
The University of Western Ontario
London, Ontario, Canada

Abstract. It is becoming increasingly prevalent in research towards automatic music composition to make use of musical information extracted and retrieved from existing music compositions. Unfortunately, this can be an unnecessarily complex and tedious process, given the incompatibilities in music modeling, organization, and representation between extraction and composition algorithms. This paper introduces the Musical Data Organization platform and framework (MUSIDO), which is aimed at resolving this problem by providing middleware to facilitate access to musical information in a simple and straightforward fashion. In doing so, MUSIDO provides an effective method to support automatic music composition based on existing music as source data. This paper discusses MUSIDO's design and implementation, and presents our experiences with using MUSIDO to date.

1. Introduction

Automatic music composition is the process of writing music withdrawn from human intervention. Many recent efforts to improve this process focus on learning from music written by humans, which involves extracting musical features and feeding these data directly into composition algorithms. The capabilities of these approaches are strongly dependent on the information gathering mechanisms employed; fortunately, algorithms that can automate this process are becoming increasingly sophisticated [4].

In practice, unfortunately, it is difficult for composition algorithms to take advantage of multiple extraction algorithms due to differences in how these algorithms model musical data. Most composition algorithms require musical data to be carefully formatted for a specific purpose and so researchers typically prefer to format their source data in a fashion that specifically serves the needs of their own approach. Extraction algorithms tend to collect and present information in an ad hoc fashion as well [5], and the information gathered is seldom meant for use in automatic composition systems, leading to issues in interoperability.

This is particularly troublesome since most extraction algorithms tend to focus on one aspect of music for recognition and extraction, and most composition systems typically require multiple sources of information to use for composition. Integration is therefore a significant problem, because it is not just one source to interface with, but rather many. To address this issue, a unified way of communicating musical information between extraction approaches and automatic composition algorithms is needed.

Our current work introduces the Musical Data Organization platform and framework, collectively referred to as MUSIDO. The purpose of MUSIDO is to facilitate the development of automatic music composition systems that rely on existing music as source data. This is accomplished by introducing a middleware entity to transfer musical information between extraction and composition algorithms in an easy, straightforward, and flexible fashion. This middleware is comprised of two important aspects: a clearly-defined data model specifically designed to organize musical data and metadata, and a standard and supportive application programming interface (API) to provide input/output access to a platform conforming to the data model.

While other platforms and frameworks already exist for working with musical data [1,2,7,9,11,12,14,15], these approaches either lack elements useful for automated extraction or composition, or were simply not designed specifically for these tasks. As a result, they fall short in terms of their ability to specify, represent, manipulate, store, and query musical information or in terms of their programming interface for constructing extraction or composition systems. In some cases, no programming interface or ability is provided at all, meaning that a considerable amount of development effort is required to make use of them.

This paper presents the findings of our current work, examining the design and development of MUSIDO in detail and discussing our experiences in using it in developing an automatic music composition system that makes use of music extraction algorithms. We have found that MUSIDO greatly facilitates the development of such systems, and does so in a way that is a considerable improvement over existing work, demonstrating significant potential for the future.

2. Overview of MUSIDO

Traditionally, constructing an automated music composition system that uses existing music as source data encounters a compatibility issue in which the music information extraction and retrieval and algorithms and composition algorithms were not initially designed or intended to work with one another. This is shown in the left side of Figure 1, in which an incompatible Automatic Music Information Retrieval (AMIR) algorithm attempts to interface with an Automatic Music Composition (AMC) algorithm.

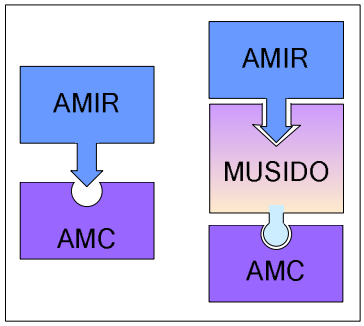

Figure 1: Traditional (left) Versus MUSIDO (right) Approaches

The main goal of MUSIDO is to resolve this issue, by providing a framework to facilitate such algorithms working together, as shown in the right side of Figure 1. This is discussed in the next section in detail.

2.1. The MUSIDO Framework

In designing the MUSIDO framework, we set out to achieve the following:

- Provisioning of a middleware entity, the MUSIDO platform, whose purpose is to communicate musical information between music extraction and music composition systems.
- Reduction of integration effort required through the support of multiple types and formats of data.
- Architectural simplicity, lowering the barrier to adoption.

This led to the development of the MUSIDO framework as depicted in Figure 2, below.

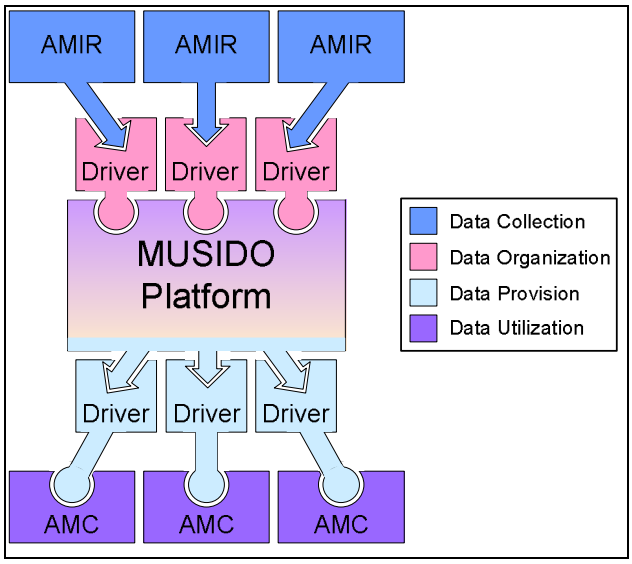

Figure 2: The MUSIDO Framework

The framework shown in Figure 2 is comprised of three types of entities, specifically algorithms, drivers, and a central platform. This contrasts existing frameworks for automatic composition, such as those involving a human component, or those resembling the left side of Figure 1, where composition algorithms make direct use of an extraction algorithm for the collection of musical data. The framework can also be viewed as a workflow with each possible entity participating in one of four separate stages:

- *Data collection*, comprised of one or more extraction and retrieval algorithms, gathering musical data from potentially multiple sources of music.
- *Data organization*, comprised of two entities: input drivers, and the MUSIDO platform, which are collectively responsible for the conversion and management of data.
- *Data provision*, comprised of the MUSIDO platform as well as drivers that serve composition algorithms.
- *Data utilization*, comprised of one or more composition algorithms for generating music.

The input to this workflow is pre-existing musical data; while we consider human compositions as the primary source, our framework does not exclude the use of synthetic data as well. Systems conforming to the framework take these data and apply each of the four stages in turn, eventually outputting newly-composed music. It is important to recognize that our framework does not provide guarantees as to the quality of music produced; this would be impossible, since quality is based on the particular entities participating (most importantly, the composition algorithm) rather than the framework itself. Instead, we strive to support automatic composition systems in order to provide the best opportunity for quality to be evaluated and reliably achieved.

2.1.1. MUSIDO Platform

The central and most significant component of our framework is the MUSIDO platform. Unlike the framework itself which exists as a conceptual entity, the platform has been implemented and exists as usable software. It is comprised of two important aspects: a flexible data model, and a software API. The platform acts as a broker of musical information, providing automated composition algorithms with musical data in a clearly-stated manner. To achieve this level of interoperability, our platform is not designed around any one particular digital music format. Rather, our data model is specifically designed to allow different types of musical formatting to co-exist, with one important caveat: only those musical features pertinent to the majority of compositions systems are directly supported in our work. This helps to achieve a concisely focused API; we leave extensions to this data model up to the individual systems that require them. Thus, improvements can be easily made in rare cases when they are needed; indeed it is not possible for any data model to represent all types of musical data and metadata, as there are a vast number of ways to analyze music itself.

Any composition system that conforms to the MUSIDO framework is one that uses the MUSIDO platform to store and organize passages of music that serve as input to the composition process. Due to its emphasis and complexity, we separately discuss the data model and API of this platform in Sections 2.2 and 2.3 respectively.

2.1.2. Drivers

The MUSIDO platform was introduced to allow music extraction and composition algorithms to work together harmoniously; however, the platform itself is comprised of a single input/output API. Newly designed algorithms will not have difficulty working within our framework, since the API for storing and retrieving data through the platform entity is well known ahead of time. Unfortunately, this is not the case for existing algorithms that have not been designed to cooperate with the platform; thus, the platform alone does not solve the problem of non-interoperability between these entities.

In order for previously created algorithms to communicate with each other through the platform, we need to introduce another entity into our framework, namely *drivers*. The responsibility of a driver is to facilitate the exchange of information between an extraction or composition algorithm and the MUSIDO platform; specifically, drivers translate information between the platform's API and the musical format expected by these algorithms. In simple terms, drivers extend existing algorithms so that they are compatible with the input/output API of our platform; in cases where these algorithms have been explicitly designed to work with MUSIDO, drivers may be unnecessary—in reality, they have already been built into the algorithm itself.

The very requirement for driver entities raises an important question: how does the MUSIDO framework differ from a traditional framework, considering that drivers could be written to allow direct communication between a music extraction process and a composition process? It is true that current composition systems which make use of existing musical data already make use of some form of driver; however, these drivers are almost always integrated directly into the composition algorithm rather than existing as a separate entity, thus demonstrating the improved modularity of our approach. More importantly, even if these drivers were extracted into their own module, the MUSIDO framework would have two distinct advantages. The first advantage of our framework is that its platform is capable of converting many types of musical data automatically, whereas extraction and composition algorithms traditionally are not. As a result, the process of creating a driver to sit between two of these algorithms would often be significantly more complex, since the drivers themselves would require complex conversion logic.

A second and even more important advantage that our framework provides is reusability of algorithms. While it may not be unreasonable for a single driver to be written within a traditional framework, such a driver would only be useful to the specific extraction/composition algorithm combination that it bridges. A problem would occur, say, if the composition algorithm designers wished to make use of a different or new extraction algorithm for gathering data. In such a case, an entirely new driver (with a different set of potentially complex conversion logic) would be needed. Thus, combining the MUSIDO platform with supporting drivers allows one algorithm/driver combination to serve a potentially limitless set of other algorithms interacting with the platform.

Mathematically, suppose we have m extraction algorithms and n composition algorithms. To interface each extraction algorithm with each composition algorithm directly would require $m*n$ points of integration. To interface each extraction algorithm with each composition algorithm using MUSIDO, however, would only require $m+n$ points of integration, to produce the driver for each algorithm in question to interface with the MUSIDO platform. Furthermore, in this scenario, if a new extraction algorithm was created, n points of integration would be required to directly interface it with each composition algorithm, and if a new composition algorithm was created, m points of integration would be required to directly interface it with each extraction algorithm. Using MUSDIO, however, only one integration step would be required with each new algorithm, to develop a driver to enable interactions with MUSIDO.

2.2. Platform Data Model

Prior to developing a software implementation of the MUSIDO platform, our work focused on the design of a data model that would satisfy the key goals of our middleware, most of which came from our review of existing algorithms as outlined in [3]. Currently we offer support for many features found in common Western civilization music; our model does not *fully* support other musical styles from different cultures, however we have preserved flexibility throughout the design process, and in some cases we have explicitly included facilities that allow alternative music forms to be expressed.

Our data model serves to organize two types of musical information. First, many elements that have some corresponding representation on a musical score (such as notes, phrases, and bars) are modeled in a hierarchical fashion. The second type of data is concerned with descriptive non-score elements (including metadata), which serves to represent musical information in a variety of capacities. In each of these cases, the data model is only meant to encompass those data types that are immediately applicable to automatic music composition, to avoid adding unnecessary complexity to the platform when compared to improved utility.

For brevity, the subsections below only provide an overview of various elements from our data model. For further details on the entire data model, the reader is urged to consult [3].

2.2.1 High-level Structure

The highest abstraction of our data model is concerned with how musical information collected by musical information extraction algorithms can be grouped and stored as individual entities, shown in Figure 3. Broadly speaking, all musical information within our platform is stored within a Repository. Repositories are analogous to database instantiations such that each Repository is capable of storing any number of Record objects; the purpose of this is to allow a way to group common Records into one conceptual set.

Record objects exist to track a single passage of music; usually this will be a song. Each can be described using the various types of data in our model, though very few data are required and in many cases fields can be left unspecified. Furthermore, Records stored in the same Repository may contain different degrees of information; there is no requirement that these Records must be similar in structure, allowing Repositories to be very flexible in organization. For example, suppose that a Mozart Repository is created, and two extraction algorithms are used to store Records in the Repository. One of these algorithms may be responsible for identifying chord progressions that occur within a set of pieces, while another algorithm may be responsible for creating a Record to supply statistics on the distribution of Mozart's use of pitch intervals. These data can be stored in separate Records, all within the same Repository, even though each Record is responsible for a different type of data than the others. In this case, Records would not be representing songs, but rather aspects of music.

In general, Records are meant to contain only those data needed for any particular application. Each of the possible data types that can be stored directly into a Record object is explained in the subsections that follow:

- A list of Sections that occur within the Record, such as verses and choruses.
- A list of musical Parts.
- A set of directives, which provide instructions on how the passage should be performed.
- A set of properties (or details), such as a title for the record or the name of a composer.

Records are flexible for storing musical information, being able to contain data from merely a few musical statistics on a piece, to entire pieces themselves. Ultimately, extraction algorithms decide what musical information should be extracted from existing music and stored. In some cases, these algorithms may extract and store important musical features along with the entire piece from which those features are derived, all within the same Record. This is possible due to the flagging system that our data model provides where themes, repeating sequences, melodic lines, and other passage types can be identified as a subset within a Record.

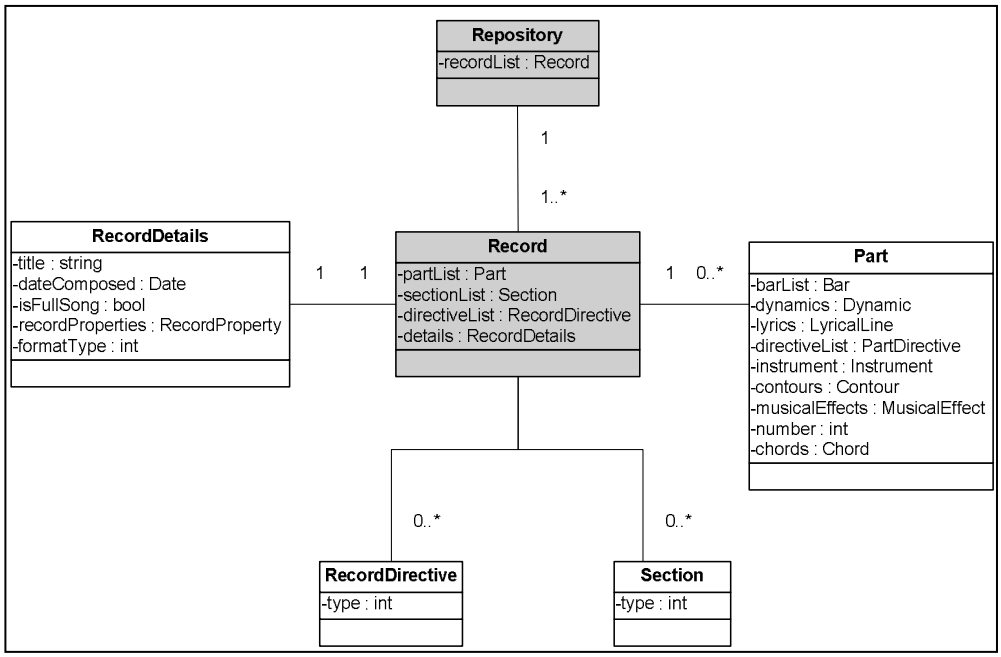

Figure 3: High-level Structure of the MUSIDO Data Model

2.2.2. Sections

We define musical structure as a set of abstract conceptual divisions within a piece, using a collection of Section elements in our data model. Each Section is a segment of music bound by a starting bar and beat and an ending bar and beat. This allows several different types of Sections to be specified for a single passage of music, with each potentially overlapping as well. As an example, a Record that represents a folk song may have one Chorus Section and five Verse Sections. This is useful for automatic composition systems to apply analysis on different parts of a piece, or to determine how the structure of newly-formed music should occur if attempting to mimic existing work.

Our model defines five types of Sections by default: Chorus, Verse, Bridge, Climax, and Alternate Ending. The model, as mentioned earlier, is highly extensible, allowing other Section types to be defined as necessary.

2.2.3. Parts

One of the most important aspects of music is the actual notes that are played. In our platform, musical Notes and Bars are assembled into Parts. This follows common conventions: most prominent digital music formats organize data this way as well, including GUIDO [8], MuseData [9], MusicXML [6], and Lilypond [13], since printed musical score is organized in this manner. Platform Records contain a set of Parts, each of which contains the following: a unique integer identifier number, an Instrument, a set of directives, a set of Musical Effects, a set of Lyrics, a set of Dynamics, a set of Bars (containing Notes in Note Positions), a set of Contours, and a set of Chords (serving as a Chord progression).

Some of these associations may be somewhat obvious in design. It is clear that Bars should be associated with Parts, according to the common layout of music. The integer assigned to a Part serves as a unique ID value within a Record; this allows data associated with the part to be retrieved directly, or through the data projection system supplied by the platform. Also, the association of an Instrument to a Part merely reflects the same common association that exists in real-world music; both written score and digital formats such as MIDI already assign single Instruments to Parts, even though it is possible that two instruments could potentially have notes on the same musical line, as is sometimes done in choral music. In such cases, our platform's 1:1 mapping of Part to Instrument requires that polyphonic multi-voiced parts be separated into separate lines of music. As a practical aide to most existing algorithms that deal with MIDI, our platform implementation includes a subclass of Instrument specifically designed to represent a MIDI instrument, including a field for a MIDI patch number.

In contrast to the more obvious design decisions above, it may be less apparent as to why the remaining elements are associated with Parts, rather than some other entity. For example, many types of musical contours can exist at a finer granularity, such as pitch contours within Bar objects. It would seem advantageous to associate these Contours with Bars rather than with Parts; in the case where a Contour is needed that spans an entire Part, one could be created from the individual Contours of each Bar contained within, although some method of determining Contour symbols between Bars themselves would be needed. While this is true, the MUSIDO platform (and Records in particular) is designed to hold partial information. Consider a music extraction algorithm that operates on MIDI data by extracting a pitch contour from the notes that occur in a melodic line. If our platform required that the contour be broken down on a per-Bar granularity, then some understanding of the Bar divisions of the piece would be required. Unfortunately, MIDI data does not include any notion of Bar lines or divisions; thus, the algorithm would not be able to specify the contour within the platform unless it first applied some additional extraction approach to Bar induction; certainly a considerable problem, and even more so in other cases such as Chord progressions, where the challenge of extracting Chords is already very difficult. Thus, our approach allows the full specification of a Contour within a Part, regardless of the presence of musical Bars.

In some cases it may make sense to track a type of data on a Bar-per-Bar basis, and yet this same type of data may also cross

bar line boundaries. For example, many Dynamics can easily be attributed to a specific note within a specific Bar. Despite this, Dynamics may also span across several Bars; for example, a crescendo may occur over a long sequence of notes. To store these data, two solutions exist: track Dynamics within Bars with complex analysis required for those Dynamics that cross bar lines, or simply track Dynamics according to Parts, requiring that a start time and possibly a finish time be set. The latter option maintains simplicity, particularly for developing a platform API, whereas the former requires complex and expensive algorithms to coordinate dynamics participating in several Bars. Thus, we take the latter approach, applying this to Lyrics, Chord progressions, and Musical Effects also.

2.2.4. Directives

The purpose of a directive is to provide general musical instructions for the entities that it is associated with. In our platform, we make use of two types of directives.

The Record Directive type may seem to serve a similar purpose to the Record Details type (discussed in next section); while both provide data that affects or describes the entire Record as a whole, they are different in two regards. First, Record Directives describe musical features while Record Details are responsible for musical metadata. In particular, Record Directives describe how music should *sound*; our current platform includes a core set of directives, while extensions to these may be included by other specific systems. The second difference between these entities is that Record Directives take place at a specific point within a piece, rather than applying to the piece universally. This is reflected in the hierarchy of the class; each directive has a distinct start and end point as defined by the four temporal variables tracking bar and beat timing.

Part Directives closely resemble Record Directives; their purpose is very similar, only limited to the scope of the particular Part that they are associated with. A wider range of directive types exist for Parts; in particular, data concerning how a piece is traversed and what play style should be used are included, again leaving the option for additional directives to be defined by individual systems using our framework.

2.2.5. Record Details

Aside from musical features, Records also contain metadata describing this information. There is potentially a large amount of data that could exist on a musical piece or passage; this was learned early in the data modeling process of the Record class, where the number of variables and functions involved quickly became unreasonable to track in a concise manner. Thus, we have chosen to separate all metadata described for musical Records in a separate object called Record Details. Record Details can include a wide variety of attributes, including title, composer, composition date, genre, status, quality ranking, data format, and so on.

There are two purposes for storing Record metadata in general. First, these data may be useful in describing some aspect of music to be considered by a composition system when generating new compositions; for example, a composition system may use the title of an existing piece to give some indication of what a newly-composed piece should be named. The number of applications in this regard are probably few, but still potentially useful. The second and likely more common way to use these metadata is for the purpose of Record organization. For example, a composition system using a large Repository of Records may wish to isolate only Jazz Records, or perhaps only those musical passages written by a particular composer. This descriptive information lends well for categorizing Records into utility-specific groups.

2.3. Application Programming Interface

The API supported by the MUSIDO function consists of three categories of functionality: input, output, and processing. Input and output functions are fairly straightforward: input functions take musical data into the platform, while output functions allow musical data to be retrieved, either at a low-level in a more direct fashion, or at a high-level, in which the data can be organized to make common information-gathering tasks simpler to accomplish.

Processing functions are provided to enable computational offloading. In some cases, the data stored in MUSIDO may require analysis or manipulation that goes beyond mere reorganization. We have therefore included some functions that act as composition services, processing common tasks that many composition systems require based on our observations as discussed earlier in this paper. These include functions for format conversion, contour conversion, data projection, transposition, statistical calculations, and so on.

Further details on this API can be found in [3].

3. Prototype Implementation

The primary goal of the MUSIDO platform is to allow a wide range of music extraction and composition algorithms to communicate information across the platform. In an effort to satisfy this goal, we have chosen to implement this middleware in two different languages: Microsoft's J# .NET, and Sun's Java. These languages are designed to operate over a diverse range of heterogeneous operating systems; while Java currently offers greater portability, an increasing number of software developers are making use of the .NET platform, and we felt it was important to provide application interoperability in this environment as well. Our selection of J#, rather than another .NET language, was motivated by the potential for simple code migration. J# uses a syntax identical to that of Java, and nearly all of J#'s API is a subset of Java's; precisely, J# is equivalent to J2SE v1.1.4. As a result, our code was written in J# knowing that migration to Java would be trivial. Also, due to the .NET framework's internal language compatibility, our platform can be compiled as a .NET dynamic link library (DLL) and used in other more common .NET languages, such as C#, C++, and Visual Basic.

Unfortunately, the J2SE v1.1.4 API, and thus J#'s API, is not sufficient to allow many automatic composition systems to be written in J#. The source of this problem is a lack of support for MIDI data; specifically, the *javax.sound.midi* package was not added to J2SE until revision 1.2, and so this package is not available when using J#. As previously mentioned, compositions systems frequently use MIDI data to specify newly-written music, and so the absence of complex MIDI-creation logic serves as a major deterrent to the use of J# for a system conforming to the MUSIDO framework. Currently, the best way to avoid this problem is to refrain from using J# as a potential language for developing a music composition or extraction system. The popular language C# does include MIDI data support, and the MUSIDO platform can still be used in this language when imported as a DLL package.

For developers desiring to use J♯ as a basis for their systems, an alternative does exist; in fact, the AMEE™ composition system [10], with which most of our validation was done, is written in J♯. AMEE™ uses a customized J♯ version of the MIDI package from Java. This package was originally taken from the GNU Classpath project, and was modified to work in the .NET environment by the developers of AMEE™. This solution not only allows some composition systems to work with MIDI in J♯, but it also preserves the code migration benefits of working with Java/J♯ syntax. Thus, while this customized package is not part of our platform's design, we consider it an important extension for systems using MUSIDO in J♯.

4. Experiences

To validate MUSIDO's use in supporting automatic music composition, we conducted a variety of tests and experiments. This section provides a brief overview of our experiences; the reader is urged to consult [3] for additional details as necessary.

Initial testing involved ensuring that MUSIDO could effectively communicate musical information between music extraction and composition algorithms. Through the use of simple algorithms, drivers were constructed to interface them with MUSIDO, and they were able to use MUSIDO easily and efficiently through its API to store, organize, and retrieve various musical elements.

More substantial evaluation of MUSIDO came in the form of testing with the automatic music composition system known as the Algorithmic Music Evolution Engine (AMEE™), as discussed earlier. Four simple music extraction algorithms were developed to gather themes, chords, probabilistic values, and contours from existing pieces of MIDI music. This data was fed into MUSIDO, and accessed by AMEE™ through the use of appropriate MUSIDO driver modules.

In doing this integration, we found that AMEE™ was able to produce music quite effectively using the musical information retrieved from MUSIDO, and was able to produce music using MUSIDO that was identical to a direct integration with the aforementioned extraction algorithms. The music composed in both cases was identical, both in the time, onset, and rhythmic feature domains. This helps to confirm the correctness of the MUSIDO platform itself; no loss of information on pitch, rhythm, or onset time is experienced when the platform is used. In the process, we did find, however, that MUSIDO greatly facilitated integration efforts, allowing AMEE™ to access musical information far more easily than with a traditional, direct integration with the algorithms.

In the end, our initial experiences indicated that MUSIDO both allows for an efficient and effective exchange of musical information between music extraction and composition systems, and enables this exchange far more readily than direct integration. This demonstrates that MUSIDO is quite promising for supporting automatic music composition efforts in the future.

5. Conclusions and Future Work

Using existing music as source data in automatic music composition systems is an approach attracting more attention from the research community. Our current work in MUSIDO is aimed at supporting this work by providing a framework to allow music extraction and music compositions to exchange musical information to support composition processes. Early experiences with MUSIDO have been quite successful, demonstrating great promise for continued work in the future.

There are several potential research directions to be explored in the future. We would like to continue experimentation with MUSIDO, in particular through user evaluation of MUSIDO and its results. Support for additional music formats and representation could be added to MUSIDO, allowing the platform to work with music in GUIDO notation, MusicXML, and sampled formats. We would also like to study the extension of MUSIDO to include musical concepts that are currently unsupported, including musical elements from other cultures.

References

[1] C. Agon, G. Assayag, M. Laurson, and C. Rueda. Computer Assisted Composition at Ircam: PatchWork & OpenMusic. *Computer Music Journal*, 23(5), 1999.

[2] G. Assayag, M. Castellengo, and C. Malherbe. Functional Integration of Complex Instrumental Sounds in Music Writing. *In Proceedings of the 1998 International Computer Music Conference*, San Francisco: International Computer Music Association, 1985.

[3] R. Demopoulos. Towards an Integrated Automatic Music Composition Framework. *Masters Thesis*, Department of Computer Science, The University of Western Ontario, London, Canada, May 2007.

[4] R. Demopoulos and M. Katchabaw. Music Information Retrieval: A Survey of Issues and Approaches. *Technical Report #677*, Department of Computer Science, The University of Western Ontario, London, Canada, January 2007.

[5] S. Doraisamy and S. Rüger. A Comparative and Fault-tolerance Study of the Use of *n*-grams with Polyphonic Music. *In 3rd International Symposium on Music Information Retrieval*, Paris, France, October 2002.

[6] M. Good. MusicXML for Notation and Analysis. *In The Virtual Score: Representation, Retrieval, Restoration*, W. Hewlett and E. Selfridge-Field, eds., MIT Press, Cambridge, MA, 2001.

[7] M. Henz, S. Lauer, and D. Zimmermann. COMPOzE: Intention-based Music Composition through Constraint Programming. *In Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, Toulouse, France, November 1996.

[8] H. Hoos, K. Hamel, K. Renz, and J. Kilian. The GUIDO Notation Format – A Novel Approach for Adequately Representing Score-Level Music. *In Proceedings of the International Computer Music Conference*, 1998.

[9] W. Hewlett. MuseData: Multipurpose Representation. In *Beyond Midi: the Handbook of Musical Codes*, MIT Press, Cambridge, 1997.

[10] M. Hoeberechts, R. Demopoulos, and M. Katchabaw. A Flexible Music Composition Engine. *Proceedings of Audio Mostly 2007: The Second Conference on Interaction with Sound.* Ilmenau, Germany, September 2007.

[11] D. Huron. The Humdrum Toolkit: Reference Manual. *Center for Computer Assisted Research in the Humanities*, Menlo Park, California, ISBN 0-936943-10-6, 1995.

[12] M. Laurson and J. Duthen. PatchWork, a Graphical Language in PreForm. *In Proceedings of the International Computer Music Conference*, San Francisco, CA, 1989.

[13] H. Nienhuys and J. Nieuwenhuizen. Lilypond, a System for Automated Music Engraving. *In XIV Colloquium on Musical Informatics*, Firenze, Italy, May 2003.

[14] D. Psenicka. FOMUS: A Computer Music Notation Tool, *FOMUS Project Documentation*. February, 2007.

[15] A. Sorensen and A. Brown. Introducing jMusic. *In InterFACES: Proceedings of The Australasian Computer Music Conference*. Brisbane, Australia. 2000.