DECLARATIVE MUSIC COMPOSITION WITH EVENT GRAPH TRANSFORMATIONS

Taosheng Qiu
National Institute of Informatics, Japan
asrcpq@gmail.com

Ryutaro Ichise
Institute of Science Tokyo, Japan
ichise@iee.e.titech.ac.jp

ABSTRACT

Representing music in declarative languages allows accurate, human-readable representation, and the automated processing of musical data. Many existing music programming languages, like engraving systems, focus on high-level representation of musical notation, offering users limited access to low-level event processing. Other music programming languages, like music processing libraries, focus on processing low-level events, thus cannot directly be used to declaratively represent musical concepts.

In this work, we propose a unified graph-based event model used throughout the compilation process, from high-level musical notations to low-level audio rendering. Musical events are represented as graph nodes connected by edges representing temporal dependencies, offering a more expressive model than traditional hierarchical series-parallel models. Event graph transformations are explicitly defined as first-class functions, which give users full transparency and extensibility to the compilation procedure. This framework is implemented in a general-purpose language that offers flexible syntax for convenient music writing, without introducing language-specific biases towards any musical system. The expressivity, extensibility, and flexibility advantages of our system are demonstrated through several examples of music segments and their implementations, involving complex temporal voice structures and notations that previous systems struggle to handle.

1. INTRODUCTION

Music representation languages use programming languages to represent and process music. Compared with interactive editors, music programming languages can represent musical notations declaratively, and enable templating musical structures or batch-editing of objects.

A core design problem of music programming systems is the framework for representing and processing musical data. Traditional systems use a fixed depth structure or series-parallel graphs with a fixed list of musical objects, with associated predefined methods for compiling music directly to sheets or MIDI. As a result, music representation languages generally fall into two categories: high-level engraving systems, which offer highly specialized DSL for music notation but limited programmability; and low-level event-based systems, which offer full programmability but lack the ability to represent musical notations as declarative contexts.

To solve this problem, we propose a unified graph-based event model that use first-class transformation functions to explicitly modify its mutable event graph. Contextual information is explicitly represented as nodes propagated during compilation. Compilation is based on first-class transformation functions, which allow users to fully control the algorithm, its execution order, or creating new musical notations with customized compilation flow. The framework is implemented in a general-purpose language with runtime syntax support, allowing users to use and create syntax forms without introducing language bias for any specific musical systems. This system offers a unified approach that provides both simplicity for music composition and full extensibility for music research and analysis.

This paper is structured as follows. In Section 2, we introduce several existing music representation systems. In Section 3, we introduce the event graph representation. In Section 4, we introduce the design and algorithms of event graph transformation. In Section 5, we introduce language and library implementation of the graph event model and transformation functions. In Section 6, we provide several music representation examples to examine the advantages of our system. In Section 7, we summarize and conclude this work.

2. RELATED WORK

Traditional music representation systems focus on sheet engraving. LilyPond [1] is the most popular music engraving programming language. It is inspired from MusixTex [2] but was then separated and reimplemented in C++ and GNU Guile, a language based on scheme R5RS [3]. LilyPond offers a DSL with specialized syntax, abundant pre-defined musical notations, and a decent MIDI rendering engine. Another notable engraving language is the ABC notation [4]. Similar to LilyPond, ABC notation offers convenient syntax and rich primitives for music annotations. Additionally, MusicXML is also considered a low-level engraving language, due to its declarative representation of musical notations. These DSL focuses on direct correspondence to sheet music, thus offers limited programmability, such as extending the DSL with custom notations or algorithmically generate events to reduce repetitions.

The music21 toolkit [5] is a sheet music analyzing, searching, and transforming library. Music21 also provides a

lilypond-like DSL parser and a MIDI exporter. Internally, music21 is based on an object-oriented representation of musical objects. Therefore, it is more restricted to classical music structures, rather than providing a general music composition framework.

Some music language propose to use operators for music composition [6]. Euterpea [7] and related systems [8], [9] are haskell libraries using graph grammars for music composition. These systems are based on series-parallel graphs, i.e., graphs created from series and parallel compositions. Instead of data, these systems usually represent music as the result of evaluation, thus do not allow efficient editing of the music that have been created. Since musical notations need to be defined as procedures, they cannot be easily exported or stored, which becomes problematic for declaratively representing non-monotonic events like acciaccatura. Another related idea [10] represents musical data as small graphs called tiles, and uses several operators to combine those graphs. This work mainly proposes a mathematical model for building music as immutable graphs, but does not give detailed solutions for processing and representing musical notations.

Finally, there are low-level music event models. The most well-known low-level music representation is MIDI, with its extensions MPE and MIDI 2.0 [11]. MIDI is the standard of digital music exchange data format. A major limitation of MIDI is its channel-based controller. True polyphonic controllers are implemented by MIDI Polyphonic Expression (MPE) and MIDI 2.0. Another example is the Open Sound Control (OSC) [12], a simple protocol for communicating arbitrary events between multimedia devices. These representations focus on representing music as low-level events, thus are not suitable for written by hands.

3. EVENT MODEL

In this section, we will introduce the event graph model for representing musical objects and relations. The graph model could store the local context of musical information, to delay the evaluation from graph building stage to later transformation functions.

3.1 Event Graph

An **event graph** is a directed graph (N, E). Each event $e \in N$ contains a key-value attributes table. For example, events may have a `type` key to indicate the type of the event. Some attributes that have a list of other nodes as values are called **edge attributes**, the values give a list of nodes N such that every $n' \in N$ is connected by edges using the key as the label: $(n, \texttt{key}, n')$.

Event can be exported and loaded as a text-based format at any time, which enables flexible storage for data exchange and caching for incremental building. The exporter will assign an `ID` to each events. Each event is exported to a line containing multiple entries:

$$\texttt{id} = \#\texttt{ID}$$

$$\texttt{key}_i = \begin{cases} \#\texttt{ID}_i^1, \#\texttt{ID}_i^2, \ldots & \text{edge attributes} \\ \texttt{value}_i & \text{otherwise} \end{cases}$$

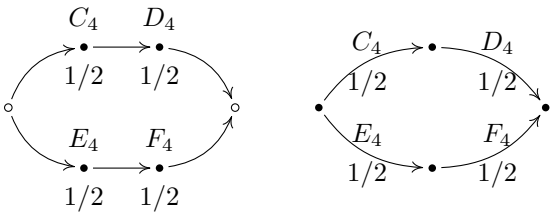


Figure 1. Our node model (left) and the corresponding edge model (right). Rational numbers show the duration of each note. White nodes in the left model are joint nodes for connecting other events.

Representing events as notes is different from some other graph-based models [10] which represent notes or rests as edges connecting anonymous vertices. An example is given in Fig. 1. Our model uses **joint** events to bridge multiple events. An advantage of representing events as nodes over edges is that nodes can be naturally extended with custom attributes to associate with other events, such as dynamic and articulation controlling. Events in an edge model are required to take a duration. Since edge models has shared nodes for different events, attributes need to be applied on the edges, which reduces the integrity and simplicity of the graph model.

An important edge attribute is called the **temporal dependency edge**.

Definition 1. *Node A is called to temporally depend on node B, if a pair of `next` and `prev` edges exist:*

$$(n_B, \texttt{next}, n_A)$$
$$(n_A, \texttt{prev}, n_B)$$

*If node X has a path of `next` edges to node Y, X is called a **temporal predecessor** of Y; Y is called a **temporal successor** of X.*

Our model, as given in left part of Fig. 1, visualizes `next` dependencies between notes as edges. The temporal dependency subgraph must be acyclic, i.e., it is not allowed to find a loop that contains only `next` or only `prev` edges. Temporal dependencies are used for defining structures of the music. Temporal dependency is created at the first stage of event graph building, before beat-based or bar-based time calculation. Most pre-defined transformation functions navigate the event graph through temporal dependency edges. For example, to reserve the duration of an acciaccatura, the temporal dependencies define notes that should be squeezed to provide the space.

3.2 Context

Many music representation languages implicity track contexts, such as octave and duration during parsing. The event graph transformation model could integrate the context handling, to make the graph building stage fully context-free. We define **Context** as data stored in some nodes that affects other nodes based on the event graph structure. For example, users can create a note with only the note name. Other

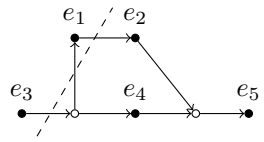
attributes of the note, such as octave, duration, time, are determined by its temporal dependency recursively.

We define **context propagation** as transformation functions that resolve context that affect temporal successors. It is often observed that temporally related notes tend to share similar attributes such as instruments. For example, in MIDI protocol, most events need channel information. Explicitly storing all context in each note as the internal data representation of music programming languages leads to scalability and flexibility problems. Instead, the graph transformation functions propagate context during compilation. The propagation procedure will temporarily construct some **context variables** that can be changed and propagated over temporal dependencies, minimizing the information that each node needs to carry during all compilation stages. Context like time and octave are propagated right after the algorithm stage, context like synthesizer parameter and volume, are kept and propagated until mixing stage,

However, problem occurs when a node is temporally dependent on multiple nodes of different values. Similar case are also observed in engraving markup languages, where inconsistent octave and duration is written before merging multiple voices or notes of a chord. For example, LilyPond [1] always uses the context of the first note. However, for graph-based representations, it will be less reasonable to force an order of edges. In our SMM system, we generally require all dependencies to hold the same value for some variable, to use that value for the current node:

Definition 2. *For any context variable V, If there exists an event subgraph $G' \subseteq G$, that all dependencies of G' in $G \setminus G'$ is an event that set V to a constant value v, while there is no node in G' that set V to a value $v' \neq v$, all nodes in G' are said to* ***naturally inherit*** *the context value v.*

An example of natural inheritance is given in Fig. 2. In this figure, e_1 and e_3 are events that set context variable d. The remaining events are note events. The only possible subgraph that contains e_5 with all dependencies setting d, but no internal events setting d is shown as the right part of the dashed line. However, due to the inconsistency of the value set by the two dependencies, the duration of e_5 cannot be naturally inherited.



e_1 $\texttt{type} = \texttt{setDuration}, \texttt{d} = 1/2$
e_2 $\texttt{type} = \texttt{note}, \texttt{note} = \mathrm{C}$
e_3 $\texttt{type} = \texttt{setDuration}, \texttt{d} = 1$
e_4 $\texttt{type} = \texttt{note}, \texttt{note} = \mathrm{G}$
e_5 $\texttt{type} = \texttt{note}, \texttt{note} = \mathrm{E}$

Figure 2. Example event graph using temporal dependency `next` as edges. Nodes right to the dashed line forms an invalid subgraph due to inconsistent duration settings between e_1 and e_3.

4. EVENT TRANSFORMATIONS

We defined the event graph as the general musical data representation model, and the context which include variables to be later propagated to other nodes. In this section, we introduce event transformation functions which handles conversions between event graph representations and context propagation. We also introduce a pipeline to show the example transformation steps for a typical environment.

4.1 Event Traversal Function

SMM library provides a iterator for templating event traversers useful for writing event graph transition functions. The traverser builder is extended from Kahn's topological sort. As shown in Algorithm 1, this algorithm takes an event graph, a context propagation function f_c, and an ordering function f_o. The context propagation function provided by the user takes the current node and a list of contexts of its temporal dependencies, and returns the context of the current node. The ordering function takes a node and returns a value as a tie-breaker for the topological sort. The procedure returns an iterator to provide the sorted nodes and context.

The topological sort is necessary even after the time of all events have been calculated. To represent events in perfect timing, the order of the events with the same value given by the ordering function must be distinguished by the topological order. For example, if a velocity change event occurs at the same time as a note event, then the topological order will be necessary to decide whether the velocity will be applied to that note event.

A direct application of this function is to implement preview rendering in SMM, by using the time attribute as the ordering function: $f_o(n) \coloneqq n[\mathrm{t}]$, and propagating the renderer function, which takes a note event and returns the rendered audio buffer, as the context. For audio streaming, the framework of SMM is based on rendering each notes independently. This is different from traditional realtime userspace audio drivers such as JACK [13] based on streaming tiny audio buffers between audio processing modules. In SMM, Audio or event streaming can be created from the traverser builder by using the time attribute as the ordering function.

4.2 Reference Transformation Pipeline

This event graph model is used throughout the compilation of the music. The compilation procedure is controlled by transformation of events programmed by users. We will introduce a reference pipeline of our model in four stages, by aligning the compilation procedure with some common musical data formats:

Declarative stage. The high-level algorithmic representation of music, represented as the evaluation result of initial graph building. The compilation of this stage evaluate templates to flattened note-based structure, then propagate necessary context, like octave and duration, to each note. Note that notes in this stage are considered to correspond to a note mark on the music sheet, not the actually played notes.

Algorithm 1 Kahn's Algorithm with Context Propagation

Require: Input event graph G
  A context propagation function f_c.
  An ordering function f_o.
Ensure: An iterator of ordered events and contexts.

Initialize $H \leftarrow$ empty min-heap.
Initialize $Contexts \leftarrow$ empty map.
Initialize $Indegree$ map for nodes in G.
for each node u with $indegree[u] = 0$ **do**
  $Contexts[u] \leftarrow f_c(u, \{\})$
  H.push$((f_o(u), u))$
                ▷ Initialize H with custom order
end for
while H is not empty **do**
  $f_o(u), u \leftarrow H$.pop()
  $P \leftarrow \{Contexts[p] \mid p \in u[\texttt{prev}]\}$
  $Contexts[u] \leftarrow f_c(u, P)$
                ▷ Compute and store context
  yield $(u, Contexts[u])$
  for each $v \in u[\texttt{next}]$ **do**
    $Indegree[v] \leftarrow Indegree[v] - 1$
    if $Indegree[v] = 0$ **then**
      H.push$((f_o(v), v))$
    end if
  end for
end while

Therefore, it does not require every individually played note to be explicitly represented. For example, music notations like glissando and vibrato produce more note events than the sheet notation, but is not expanded at this stage.

Sheet stage. The high-level representation of musical objects. This stage can be used for exchanging data with symbolic musical formats like engraving systems. The compilation of this stage includes interpreting musical notations, and resolve the time to absolute beat for each note.

Performance stage. The low-level representation of musical events. This stage can be used for exchanging data with musical communication protocols like MIDI. The compilation of this stage includes frequency calculation, time and tempo calculation, and note rendering.

Rendering stage. The rendered audio. This stage can be used for exporting or streaming rendered audio. The compilation of this stage includes audio buffer mixing and data streaming or exporting.

A key design difference of SMM is that all transformations are based on modifying a mutable event graph. Most previous systems are based on constructing immutable tree-like representation in a single pass, by carefully tracking multiple user-customizable contexts during parsing a specialized DSL. However, representating musical objects as data, instead of as result of evaluation, is required for representing non-monotonic musical objects, like acciaccatura. Additionally, storing musical objects as mutable graphs allows the separation of stages for data exchange with various formats.

5. IMPLEMENTATION

A flexible language is required for efficiently and universally building event graphs and writing transformation functions of event graphs introduced in previous sections. In this section, we introduce the implementation of the `pv` language, which includes a language which allows user-defined runtime syntax, and the S-expression Music Markup (SMM) music library implemented on top of it.

5.1 Base Language

We propose a general-purpose language `pv` that can be used for writing graph creation and transformation functions. The syntax of the language is close to Scheme, which is based on S-expressions, and rely on lambda functions instead of object-oriented modeling. The interpreter is currently implemented on top of Python, allowing interoperability between Python and underlying FFI libraries. Unlike Python, it allows defining runtime syntax with unrestricted expressivity through vau expressions [14]. `pv` also provides a simple module system based on relative path of the current source file, which enables dynamic loading of musical representations.

One major advantage of using a general language over a DSL is that SMM has no builtin keywords or specialized syntax for musical representation. All musical concepts are implemented as library functions and syntaxes, without polluting the syntax or the namespace. For example, LilyPond represents note names like `c`, cis. ABC notation represents note names like `c`, `^c`. In SMM, the default graph builder's syntax parses some strings like `c`, cs, but user can create custom syntax to simulate other notations, or to implement notation systems for other temperaments or simulating existing notations. Meanwhile, those name can be still used as variable and function names. Note names will override variable names, only in vau functions that customize the evaluation order.

Although libraries like music21 [5] also provide convenient notation systems, they are implemented as a DSL on top of the host language, thus inherently have the same problem as those engraving languages. Such subsystems need to be explicitly parsed and quoted, which increases the verbosity of music writing. Programming on such DSLs relies on manually templating the DSL, offseting the advantage of using a general library for music representation. For example, users cannot conveniently call host language functions inside the DSL, or extend the parser with custom notations, without rewriting it.

We implemented our S-expression Music Markup (SMM) system on top of `pv`, which provides a reference implementation of a set of convenient graph builder and transformation functions. Due to the scalability limitation of the Python-based interpreter, some implementations are split into Python modules loaded by the `pv` programs. For example, audio sample parsing and audio rendering are implemented in NumPy modules. Music is represented as functions, and dynamically loaded by external wrappers which define the interface. For example, a default sheet-to-audio compiler is provided in SMM library, which expect a

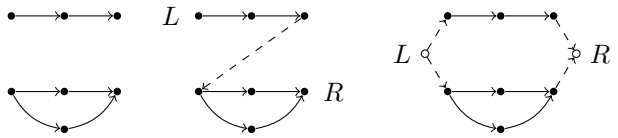


Figure 3. Two subgraphs (left) merged with series (middle) and parallel (right) builders. Dashed lines show additional temporal dependencies. Hollow circles in the parallel builder are joint nodes. Event attributes are omitted. L/R represent the output left and right nodes of the builder.

```
[fn [series meval] [return [vau [. l] env
    [if [== 0 [len l]] [panic 'emptyList]]
    [= first null]
    [= prev null]
    [forin ll l
        [= [p n] [meval ll env]]
        [if [== first null] [= first p]]
        [if prev [connect prev p]]
        [= prev n]
    ]
    [return first n]
]]]
```

Figure 4. Definition of the series operator, defined as a meta-builder that takes an `meval` function and returns the builder function with captured evaluator.

`sheet` function in the loaded module, and will render the evaluated sheet stage representation to final audio file.

Additionally, some utilities are provided. The `soundfont` project extracts SoundFont2 parameters and samples to raw samples. A MIDI parser is implemented to convert MIDI files to performance stage.

5.2 Graph Building

The default parser model in SMM is based on the classical series-parallel graph builder. Inside the builder, every function returns left and right nodes. A series combinator concatenates events one after another, and a parallel combinator fuse all left nodes and all right nodes respectively to a joint node. An example is given in Fig. 3. The series and parallel combinators expect list of pairs of left and right notes, and returns its leftmost and rightmost nodes after merging.

SMM's implementation of the series operator is given in Fig. 4. The function is a meta-builder that captures the evaluation function `meval` and returns the combinator function, which takes any amount of variables, parsing them as chain of events, then concatenate these chains consecutively.

Since during parsing the syntax needs to be recursively applied to child structures, these builders need to be created by taking the custom evaluation function. This allows different parsers to reuse the implementation of these builders, while supporting custom syntax. SMM provides a note parser `line`, which implements the default note parsing on top of the series and parallel operators. The implementation is shown in Fig. 5. The three functions: `line/call`, `line/;`, and `line/:`, are mutually recursive, that the

```
[fn [line/call l env]
    [cond [[list? l] [cond
        [[== '; [@ l 0]]
            [set l 0 'line/;]
            [return [eval l env]]
        ]
        [[== ': [@ l 0]]
            [set l 0 'line/:]
            [return [eval l env]]]
        ...
    ]] [[str? l]
        [= ret [line/evalstr l env]]
        [if [! [null? ret]] [return ret]]
    ]]
]
[= line/; [series line/call]]
[= line/: [parallel line/call]]
```

Figure 5. Part of the implementation of `line` parser's evaluator. Note that `line/;` and `line/:` are instantiated series and parallel builders.

nested call among these functions will prioritize customized parsing logic. Musical keywords like note names are parsed in `line/evalstr` thus does not pollute the namespace.

The `line/call` is then wrapped into a closure which statically captures the right nodes of each call and restores the value as the left node at the next call. This allows writing multiple staves in parallel by independently concatenating each line. Since context is stored as graph nodes, as long as the graph is concatenated, all context variables can be automatically propagated until the end of the line.

An important event parsed by the `line/evalstr` is the **label** event. Similar with MIDI metaevent markers, labels are events for locating specific positions in an event graph. In SMM, labels are not automatically indexed. Instead, the corresponding local variable will be automatically set or fetched. Three forms are parsed by the evaluator:

1. `<label` get the label, report error if undefined.
2. `>label` set the label, report error if already defined.
3. `'label` get the label if defined, set the label if undefined.

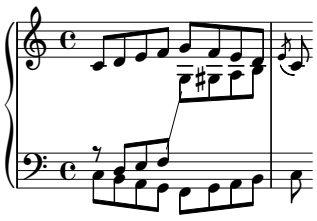
Labels can be used to merge nodes and fuse event graphs, allowing the creation of arbitrary graph structures that cannot be built with the series-parallel graph builder. Labels can also be used for locating the events, that can be passed to the later transformation functions.

6. EXAMPLES

In this section, we provide several examples of SMM. We mainly focus on comparing SMM with LilyPond, which also offers a convenient DSL built on top of Guile Scheme, a general-purpose language. Lilypond is based on a series-parallel graph model. User have limited control for the compilation from sheet to the actual performance.

Source code of the examples, the implementation of SMM, and related libraries is available online. [1] We use Alpine

[1] `https://github.com/asrcpq/2025_tenor`



```
[P1 /2 c4 d e f g f e d |
    >2 [acc [/8 e4] /2 c4]]
[P2 /2 c3 >1 b2 a g f g a b | c3]
[line/; <1 d e f g gs a b <2]
```

Figure 6. An example of splitting and merging voices that raises problem in traditional hierarchical series-parallel graph models. The line indicates that the two beams belong to the same voice. The corresponding SMM program is given below the sheet.

Linux v3.21.3 for the development and evaluation of the examples.

6.1 Expressivity of Event DAG

SMM represent music as graphs, instead of a hierarchical tree model like most other systems. This allows it to extend music representation with complex splitting and merging of voices. Consider a segment of music with three voices upper, middle, and lower voices. As shown in Fig. 6, the middle voice is split from the lower voice and is finally merged into the upper voice.

Since SMM provides series and parallel combinators for basic graph constructions, it is possible to create any DAG by connecting the graph, such as using labels introduced in Section 5.2. Traditional hierarchical music representation systems do not have a direct correspondence of this segment. Some workarounds need to be used, such as writing three independent voices to approximate the DAG model. Such misalignment between the data structure and the actual music leads to many potential problems. For example, in Fig. 6, the last note in the upper voice is an acciaccatura. Acciaccatura is defined to temporarily occupy the duration of previous notes. However, due to the misalignment, it is not possible to define the temporal dependencies of the main note, which need duration adjustments to reserve the space for the acciaccatura.

On the other hand, SMM implements accurate acciaccatura rendering, by modifying the event graph after the sheet stage. As shown in Fig. 7, in graph building stage, a special joint event is create to associate the acciaccatura with the main note as an additional temporal dependency. During beat calculation, the time of the `G_ACC` is calculated in parallel with other dependencies of the main note. In the rendering stage, the transformation function will first calculate the temporal range $t_1 \sim t_2$. Then, it traverse other dependencies of the joint event, reserving this period by adjusting the durations of the notes. Finally, `G_ACC` is inserted between the previous events and the main note. In LilyPond, most performance processing, such as rendering of grace notes, is implemented as fixed logic in the C++ backend. Therefore, this limitation cannot be solved without rebuilding the engine from the bottom.

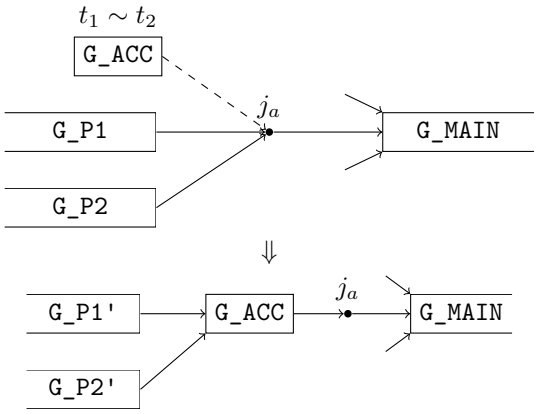
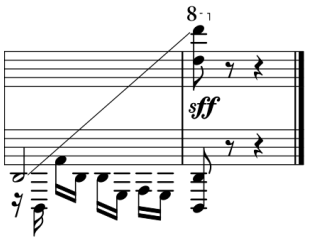


Figure 7. Visualization of transformation for rendering acciaccatura. `G_*` represents event subgraphs. j_g is a special joint node for acciaccatura which maintain an additional edge (dashed) to the ornament.




```
[P1 1/ d2 [EV gliss] c6 | /2 [: d5 d6]]
[P2 /4 r d1 a2 d d g1 a g | /2 [: d1 d2]]
```

Figure 8. Simplified last two bars of Prokofiev's Toccata in D minor. The line notates a glissando. The corresponding SMM program is given below the sheet.

6.2 Extensibility of First-class Transformations

Since SMM defines and applies transformations as first-class functions, it naturally supports extending the notation with customized interpretations. As an example, a glissando in the last two bars of Prokofiev's Toccata's is given in Fig. 8. In building stage, we denote the glissando with the starting and ending notes, with the halves whole duration of the glissando as the note duration. A declarative `gliss` event is inserted between the two notes. This example implements glissandi by specifying a target note `C6`, since the actual target note in the next bar is not involved in time calculation.

After the sheet stage, a transformation function `gliss` expands the declarative glissando to note events. The example implementation is given in Fig. 9. `pp` and `pn` give the pitch (semitone distance to `C0`). The duration `d` of each note in the glissando is computed by dividing the total duration of the glissando divided by the number of notes. A loop over a list of pitches in the glissando created new note events and build a chain between the initial two nodes.

Glissandi usually require customized algorithms for different instruments. For piano, a glissando is performed on

```
[= l [pianogliss pp pn]]
[/= d [len l]]
...
[forin pitch [slice l 1 -1]
    [= vn [% pitch 12]]
    [= vo [/ pitch 12]]
    [= n [EV note [note vn]
        [d d] [t t] [octave vo]]]
    [if [null? first] [= first n]]
    [connect last n]
    [= last n]
    [+= t d]
]
```

Figure 9. A snippet from the `gliss` transformation function which fills a chain of note events between two adjacent notes of the gliss event.

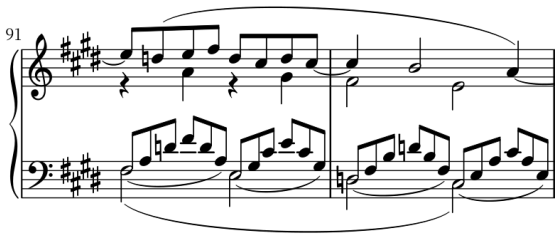
only white or black keys, processed by the `pianogliss` function, which internally generates the pitch sequence by skipping pitches that have a different key color from `pn`.

Most existing music representation systems either cannot represent glissandi declaratively, or do not provide an easy extension for handling glissando compilation. For example, LilyPond offers sheet representation of glissando, but does not render it in the MIDI output. The rendering process is often handled by internal algorithms, which can only be controlled with some exposed parameters. For many systems based on immutable graph representations, they also do not allow customized transformation functions, without inefficiently rebuilding the whole music structure. Therefore, SMM as a universal event graph models allowing transparent access to the data offers user full customizability throughout the compilation process.

6.3 Flexibility of the Language

We use an example from Debussy's first arabesque, also used in previous research [10], to show the generality of the graph model. The previous work did not give details on how to construct such graphs, which will be discussed in this section. The sheet and corresponding musical notation is given in Fig. 10. `P1` to `P3` using SMM `line` parsers are automatically concatenated between calls. For each note, an optional number follows the key name, which set the octave context variable. All numbers with a slash set the duration context variable to a rational number of beats. Note that since double-stemmed unisons are defined as multiple notes of same pitch played simultaneously, the actual interpretation such as merging these notes should be delayed to the performance stage.

In musical composition, it is usually more convenient to write all staves by grouped measures. SMM, based on runtime syntax and general-purpose graph builders, enables concise syntax for writing multiple staves or voices in interleaved units. Since context of graph building stage is explicitly stored as nodes, octave, duration, and contextual musical notations like ties in Fig. 10, are automatically continued after combining the structure. In LilyPond, it is not easy to implement a similar notation. Due to the separation of the DSL and the underlying language, the




```
[P1 /2 e d e fs d cs d cs - |]
[P2 /1 r a4 r gs |]
[P3 [: [; /3 fs a d4 fs d a3] [; 2/ fs]]
    [: [; /3 e gs cs4 e cs gs3] [; 2/ e]]|]
[P1 /1 cs 2/ b4 /1 a - |]
[P2 2/ fs e |]
[P3 [: [; /3 d fs b d4 b3 fs] [; 2/ d]]
    [: [; /3 cs e a cs4 a3 e] [; 2/ cs]]|]
```

Figure 10. Bar 91, 92 of Debussy's first arabesque and the corresponding SMM program.

```
[= T [vau [n1 n2 n3 n4] env
    [return [eval [list 'line/:
        [list 'line/; '/3 n1 n2 n3 n4 n3 n2]
        [list 'line/; '2/ n1]
    ] env]]
]]
[P3 [T fs3 a3 d4 fs4] [T e3 gs3 cs4 e4]]
[P3 [T d3 fs3 b3 d4] [T cs3 e3 a3 cs4]]
```

Figure 11. Rewriting lower staff of Fig.10 with templated pattern.

user cannot preserve the context, if the segments are parsed before concatenation. While it is possible to add an earlier preprocessing stage to syntactically concatenate the segments before parsing, it will essentially create a higher meta-DSL with Lilypond interoperability dropped.

Since the lower staff contains a pattern repeated for many bars. The pattern can be templated as shown in Fig. 11, similar to LilyPond's `define-music-function`. However, Lilypond uses a heterogeneous model for music processing and representation, which limits the flexibility of the language. Since the representation model is mainly implemented in C++ backend, users can only talk to the powerful Scheme engine with a fixed list of pre-defined musical objects.

7. CONCLUSION

In this work, we propose to use unified, mutable event graphs to represent musical data across different compilation stages. First-class transformation functions explicitly compile event graphs from high-level to low-level representation stages. We propose a general-purpose language for writing event graph construction and transformation functions. We provide several examples to show the flexibility and customizability of our systems, compared with previous musical programming languages. The first acciaccatura example shows SMM's expressivity through its unified event DAG model for music representation. The second glissando

example demonstrates the extensibility of SMM by using first-class functions for transformations, which are explicit and fully controllable by users. The third multi-voice example shows the flexibility of the language and its ability to create a convenient syntax for music writing, without adding global language biases like most music DSLs.

In summary, this paper proposes a unified framework for musical processing, focusing on generality and extensibility, and a practical implementation of the framework that demonstrates advantages in flexibility and coherence. This framework could be a useful model for musical research and declarative music programming. Since most existing declarative music programming languages are based on a direct correspondence between sheet music and their digital representations, a more general graph model with explicit transformations enables efficient and declarative experimentation with new ideas in musical research beyond traditional sheet representations.

On the other hand, SMM library is not designed to directly correspond to sheet representations, which could be a limitation in some circumstances. Currently, SMM lacks the ability to exchange data with digital sheet music formats, such as MusicXML. In the future, we plan to support visualizing event graphs in various media, such as plain text, rendered sheets, and interactive editors. However, controlling the engraving behavior of sheet music is not likely to be included in the core SMM library, due to its focus on declarative music representation. Another future work is to evaluate the usability of SMM through user studies and case studies of musical research.

8. REFERENCES

[1] H.-W. Nienhuys and J. van Nieuwenhuizen, "Lilypond, a system for automated music engraving," in *Colloquium on Musical Informatics (XIV CIM 2003)*, 2003, pp. 167–172.

[2] D. Taupin, R. Mitchell, and A. Egler, "Musixtex. using tex to write polyphonic or instrumental music," in *Proceedings of EuroTeX 93*, 1993, pp. 257–272.

[3] N. I. Adams, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, G. J. Sussman, M. Wand, and H. Abelson, "Revised5 report on the algorithmic language scheme," *ACM SIGPLAN Notice*, vol. 33, no. 9, pp. 26–76, 1998.

[4] C. Walshaw, "The abc music standard 2.1," https://abcnotation.com/wiki/abc:standard:v2.1, 2011, accessed: 2025-07-09.

[5] M. Cuthbert and C. Ariza, "Music21: A toolkit for computer-aided musicology and symbolic music data," in *Proceedings of the 11th International Society for Music Information Retrieval Conference*, 2010, pp. 637–642.

[6] D. Fober, Y. Orlarey, and S. Letz, "Scores level composition based on the guido music notation," in *Proceedings of the 2012 International Computer Music Conference*, 2012, pp. 383–386.

[7] P. Hudak, "Euterpea," https://www.euterpea.com/, 2014, accessed: 2025-07-09.

[8] D. Quick, "Kulitta: A framework for automated music composition," Ph.D. dissertation, Yale University, 2014.

[9] P. Hudak, D. Quick, M. Santolucito, and D. Winograd-Cort, "Real-time interactive music in haskell," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*, 2015, pp. 15–16.

[10] D. Janin, "A robust algebraic framework for high-level music programming," in *Proceedings of the 2nd International Conference on Technologies for Music Notation and Representation*, 2016, pp. 167–175.

[11] The MIDI Association, "MIDI 2.0 Specification," https://www.midi.org/specifications/midi-2-0-specifications, 2020, accessed: 2025-07-17.

[12] M. Wright and A. Freed, "Opensound control: a new protocol for communicating with sound synthesizers," in *International Computer Music Conference 1997. Proceedings: Thessaloniki, Hellas. 25-30 september 1997*, 1997, pp. 101–104.

[13] S. Letz, Y. Orlarey, and D. Fober, "Jack audio server for multi-processor machines," in *Proceedings of the International Computer Music Conference*, 2005, pp. 1–4.

[14] J. N. Shutt, "Fexprs as the basis of lisp function application or $vau: the ultimate abstraction," Ph.D. dissertation, Worcester Polytechnic Institute, 2010.