DETECTING NOTATIONAL ERRORS IN DIGITAL MUSIC SCORES

Léo Géré
Cnam, Cedric, Paris, France
leo.gere@lecnam.net

Nicolas Audebert
IGN, LASTIG, Saint-Mandé, France
Cnam, Cedric, Paris, France
nicolas.audebert@ign.fr

Florent Jacquemard
INRIA, Paris, France
Cnam, Cedric, Paris, France
florent.jacquemard@inria.fr

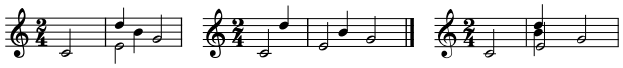
ABSTRACT

Music scores are used to precisely store music pieces for transmission and preservation. To represent and manipulate these complex objects, various formats have been tailored for different use cases. While music notation follows specific rules, digital formats usually enforce them leniently. Hence, digital music scores widely vary in quality, due to software and format specificity, conversion issues, and dubious user inputs. Problems range from minor engraving discrepancies to major notation mistakes. Yet, data quality is a major issue when dealing with musical information extraction and retrieval. We present an automated approach to detect notational errors, aiming at precisely localizing defects in scores. We identify two types of errors: *i)* rhythm/time inconsistencies in the encoding of individual musical elements, and *ii)* contextual errors, *i.e.* notation mistakes that break commonly accepted musical rules. We implement the latter using a modular state machine that can be easily extended to include rules representing the usual conventions from the common Western music notation. Finally, we apply this error-detection method to the piano score dataset ASAP [1]. We highlight that around 40% of the scores contain at least one notational error, and manually fix multiple of them to enhance the dataset's quality.

1. INTRODUCTION

Sheet music scores are, since centuries, an essential medium for the exchange of information between composers, performers, teachers, students, scholars, *etc*. We focus in particular on common Western music notation. Scores are rich and complex hierarchical objects, containing information about notes, rhythms, meter, dynamics, expression... Various encodings have been proposed over the years to represent these musical data digitally, such as **kern [2], ABC notation [3], MusicXML [4, 5], MEI [6, 7].

As complex as they can be, scores must follow a certain number of syntactic rules to be correctly analyzed, or for their content to be leveraged in different applications, including graphical rendering [8, 9, 10]. For example, a half note has twice the duration of a quarter note; notes in a same voice cannot overlap each other unless they form a chord, *i.e.* a group of simultaneous notes of the same duration. Score encoding formats are usually lax regarding compliance with these rules, which can lead to unconventional scores difficult (or even impossible) to work with. In particular, verbosity and information redundancy in these formats may be sources of inconsistencies. For example, in MusicXML, note durations are expressed both using symbols (quarter/half/eighth..., eventual dots) and numerical values (number of quarter notes on a timeline). However, the specification does not enforce any consistency between those, which can lead to issues in voice alignment and parsing difficulties. Those notational errors [1] can come from music editing software export, from conversion between different encoding formats, or from human encoders themselves. Differences in encoding have an impact on music analysis tasks [11], supporting the importance of having correct encoding to effectively leverage musical scores. Clean data is also primordial to train machine learning models [12], especially models producing complex structured output such as score data, in the context of tasks such as Automatic Music Transcription or Optical Music Recognition.



(a) MuseScore 4.5.2 (b) Dorico 6.0.22 (c) Verovio 4.2.0

Figure 1. Example of the same erroneous MusicXML file (presented in Appendix A) rendered by three different score rendering/editing software applications.

All music editing software perform validation, *i.e.* checking respect of the MusicXML specification. However, some also validate the musical content itself, looking for inconsistencies. Furthermore, programs sometimes applies automated error-fixing mechanisms, *e.g.* by raising explicit errors to the users, or sometimes by silently "correcting" the score. Yet, those automated checks are often limited and software-specific. As shown in Fig. 1, different software handles errors differently. In practice, we argue that it might be easier to point the error to the user, and let them fix it themselves rather than guessing their initial intention.

To evaluate the quality of digital score libraries containing scores from different sources, formats, and using different conventions, the GioQoso Project [13, 14, 15] proposes a quality model evaluating diverse aspects of the scores. It investigates a wide range of potential quality issues: score

[1] Composers do sometimes voluntary break some music notation rules *e.g.* for artistic reasons, but we will consider those as errors in this work.

content issues, engraving issues, metadata issues, *etc*. In this work, we only consider the *musical content* of a score independently from any engraving and graphical rendering concerns. In other words, we only deal with the logical structure of the score, without considering the exact placement of elements relative to each other on a printed sheet.

In this article, we introduce a two-step error detection method that aims at precisely localizing notational mistakes in a score. First, we perform a rhythm/time consistency check, that detects musical elements whose theoretical duration, as described by their symbol, is inconsistent with the actual duration specified in the MusicXML tags. See Section 2.1 for the description of this kind of errors and Section 3.1 for their detection. We chose to focus on MusicXML, as it is one of the most common open digital music format. In the second step, we ensure that the score complies with a set of conventional music notation rules, see Sections 2.2 and 3.2. Our validation algorithm is flexible, built upon a set of music tokens and a state machine that could easily be extended with new rules and conventions. This second step is format-agnostic and could be extended to any encoding, provided that a tokenizer is implemented for the required format. We apply this method to the ASAP piano scores dataset [1], and identify problematic sections in approximately 40% of the scores (Section 4). Our manual qualitative analysis shows that these are notational mistakes, due to annotator error and poor software exports. We also improve the overall quality of the ASAP dataset by fixing some of these errors.

2. ERROR TYPES

There are multiple sources and types of errors, and we consider the following two categories: individual errors, and contextual errors. Individual errors refer to errors on single elements that we can detect on their own, without considering their interpretation context. Contextual errors refer to elements that are correct on their own, but break musical rules when taking into account their musical surrounding, *e.g*. overlapping notes in a voice.

2.1 Individual Errors

The errors presented in this subsection are specific to MusicXML due to how this format encodes some information.

2.1.1 Duration Representation in MusicXML

In MusicXML, durations are expressed as a number of quarter notes on a timeline, represented as exact fractions. The denominator is encoded at the measure level in the integer `<divisions>` tag.[2] The numerator is stored in the individual elements, in an integer `<duration>` tag. For instance, in a measure with an attribute `<divisions>` of 24, a note with a tag `<duration>` of 48 would last the duration of $\frac{48}{24} = 2$ quarter notes. This fraction represents the actual duration of the note on the timeline, but is incomplete with respect to the symbology of the note *value*, *i.e*. whether it is a quarter note, a half note, *etc*., and whether it has augmentation dots, or belong to a tuplet. For these purposes MusicXML also includes respectively the `<type>`, `<dot>` and `<time-modification>` tags.

[2] Usually, only one single common denominator is used for the whole score, introduced in the first measure.

2.1.2 Inconsistencies in Redundant Duration Information

Individual errors often stem from inconsistencies in redundant information, typically in MusicXML's double encoding of the duration. As just explained, a note with a `<duration>` of 48 in a 24-division-based MusicXML, has a duration of two quarters notes, or a half note. Yet, the MusicXML specification does not enforce that the `<type>` tag represents a half note. Therefore, some notes can have a timeline duration that is inconsistent with their theoretical duration. Parsers then have to decide which one to use.

These inconsistencies are generally rooted in attempts to improve the user experience, *e.g*. by writing the score so that its engraving or MIDI playback matches the user's expectations – at the expense of the musical content correctness. Grace notes are a typical example when they are written as normal notes. Ornaments are another one when added as invisible notes.

Tuplets are also a source of duration inconsistencies, due to rounding errors. MusicXML theoretically allows representing any fractional duration by setting an appropriate `<divisions>` tag. Yet, some software exports scores with a default value when faced with unusual tuplets, *e.g*. 480 for MuseScore up to version 4.4. Because some notes durations cannot be expressed as a fraction with this denominator, their `<duration>` tag is rounded to the nearest integer resulting in quantization errors. For example, a note in a septuplet of eighth notes has a theoretical value of $\frac{1}{2} \times \frac{8}{7} = \frac{4}{7} \approx 0.5714$ quarter notes, while in a 480-division-based MusicXML file it would have a duration of $\frac{\text{round}(4/7 \times 480)}{480} = \frac{274}{480} \approx 0.5708$ quarter notes.

2.1.3 Dubious `<backup>` and `<forward>` Tags

MusicXML uses `<backup>` and `<forward>` tags to align the different voices on the timeline. For example, once a voice has been completed, a `<backup>` tag is used to "rewind" the timeline back to the start of the measure and start filling a new voice. Similarly, `<forward>` tags are used to artificially "fast-forward" in the measure, generally in the case of incomplete voices. Like notes, these tags have a duration indicating the displacement on the timeline.

Although there are no set rules constraining the duration of these tags, some unusual values are likely to indicate an underlying issue. Consider a 480-division-based MusicXML file, in which a `<forward>` tag has a duration of 1. This means stepping forward on the timeline by a duration of a 1024th note in a triplet, nested in a quintuplet. This is not strictly speaking forbidden, although it is unlikely. Music editing software typically uses such `<forward>`/`<backup>` tags to fill small gaps due to the accumulation of rounding errors in case of the ill-chosen `<divisions>` tags discussed above.

2.2 Contextual Errors

The errors presented so far were errors on their own and did not need any context to be qualified as errors. However,

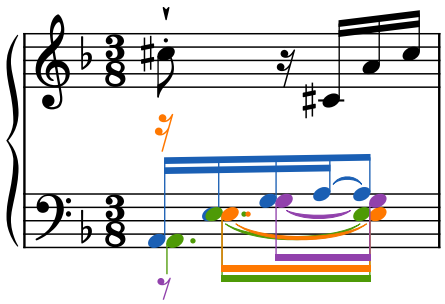


Figure 2. Example of overlapping notes in measure 356 of 3rd movement of Beethoven's Piano Sonata No. 17 in the ASAP dataset [1]. The notes are colored by voice. The notes in the green voice overlap with each other: no other note should start on that voice after the first dotted quarter note. This issue might come from a conversion error in the history of the score, merging two voices into one.

some errors are more complex and do require information about surrounding notes to be properly identified.

2.2.1 *Voice Overlap Errors*

We consider a voice to be a succession of notes, without any gap or overlap except for chords. In polyphonic music, a score can contain multiple voices in parallel, emphasizing different melodic lines. Voices can sometimes be *incomplete*, for example, they can start in the middle of the measure, along with an already started voice. In this article, we only consider *complete* voices, *i.e.* without any gaps. In case of incomplete voices, we pad them with rests.

Typical errors in voices are overlapping notes, as illustrated by Fig. 2. The second and third notes in the green voice overlap with the first one, and should have been put into a different voice.

2.2.2 *Measure Overflow*

In theory, the time signature constrains the duration of the measure, *i.e.* it sets the *nominal duration*. For example, a voice in a 4/4 measure is expected to contain the equivalent of four quarter notes. In practice, as there are valid cases in which a measure is shorter than its nominal duration, *e.g.* pickup measures or repetition bars placed in the middle of a measure. Hence, we only treat measures longer than their nominal duration as errors.

2.2.3 *Chord Errors*

Chords are groups of notes belonging to the same voice, starting at the same time and lasting the same duration. Once again, this is not constrained by the MusicXML specification. Therefore, we want to explicitly ensure that all same-voice notes starting at the same time also share the same duration, meaning that they belong to the same chord.

In addition, we can include some instrument-specific rules related to chords. In the case of the piano dataset ASAP presented in Section 4.2, the piano is not allowed to play the same note multiple times simultaneously. Hence, chords in piano scores should not contain duplicate notes. [3]

[3] Yet, there can be duplicate notes if they belong to different voices.

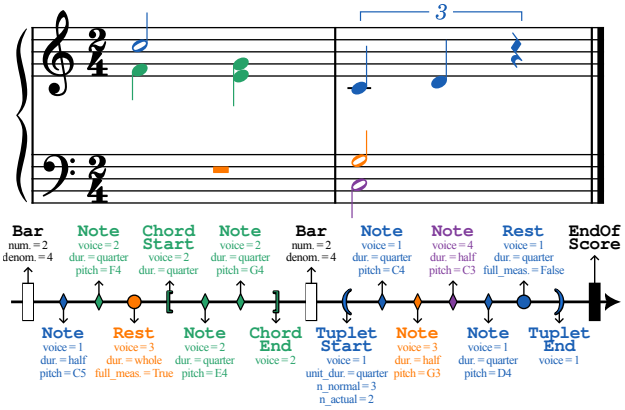


Figure 3. Example of tokenization of a score. The timeline represents the sequence of tokens. The items' shape represents the type of token, and their color the voice they belong to (only for tokens tied to a voice).

3. ERROR DETECTION METHODS

We now introduce our error detection method, which is separated into two stages. The first stage checks for individual rhythm/time consistency errors as introduced in Section 2.1, while the second one checks for contextual errors presented in Section 2.2. Individual errors are straightforward to spot as they only require analyzing each element independently. However, contextual errors are more challenging, as they require considering the neighborhood of the tested elements.

Note that we focus on verifying the actual *musical content* of the score. In other words, we *i)* consider only well-formed files, *i.e.* respecting the MusicXML specification [4], malformed files are supposed to be detected beforehand, and *ii)* exclude engraving/rendering-related issues.

3.1 Detection of Individual Errors

We ensure the duration consistency of every note and rest, as explained in Section 2.1. First, we compute the theoretical duration of each of them from the `<type>` and `<dot>` tags, taking into account any tuplet modifier tag `<time-modification>`, if present. Second, we compare this value to the actual timeline duration of the element encoded in the MusicXML. If they differ, we flag the note/rest as faulty for further inspection.

In addition to notes and rests, we also inspect `<backup>` and `<forward>` tags for suspicious durations. To do so, we ensure that each of those tags can be decomposed into a valid sequence of rests, up to some user-defined minimal duration, like 128th notes. For example, in a 480-division-based MusicXML file, a `<forward>` of duration 45 could be decomposed into a 64th rest (30) and a 128th (15) rest, but a `<forward>` of 46 could not without allowing much smaller divisions. Any tag that cannot be decomposed into an allowed sequence of rests is flagged as faulty.

3.2 Detection of Contextual Errors

Detecting more complex errors than individual errors requires taking into account the interpretation context of each

[4] `https://www.w3.org/2021/06/musicxml40`

Token type	Definition	Attributes
`Bar`	Beginning of a new measure	• $T_{\text{numerator}}$: Numerator of the time signature of the measure • $T_{\text{denominator}}$: Denominator of the time signature of the measure
`Note`	Single note	• T_{voice}: Voice index • T_{pitch}: Pitch information (step, octave, accidental) • T_{duration}: Symbolic duration (quarter note, half note, with optional dots…) • $T_{\text{is\_grace}}$: Flag indicating if it is a grace note • $T_{\text{is\_tied}}$: Flag indicating if tied to a future note
`Rest`	Single rest	• T_{voice}: Voice index • T_{duration}: Symbolic duration • $T_{\text{full\_measure}}$: Flag indicating a full-measure rest
`ChordStart`	Beginning of a chord	• T_{voice}: Voice index • T_{duration}: Symbolic duration • $T_{\text{is\_grace}}$: Flag indicating if it is a grace chord
`ChordEnd`	End of a chord	• T_{voice}: Voice index
`TupletStart`	Beginning of a tuplet	• T_{voice}: Voice index • $T_{\text{base\_unit}}$: Base symbolic unit • $T_{\text{n\_normal}}$: Number of base units it should normally take (outside context) • $T_{\text{n\_actual}}$: Number of base units it actually consists of (inside context) e.g. *for a tuplet of 3:2 eighth notes, those last three attributes are respectively "eighth", 2 and 3.* We also use the normal and actual duration of the tuplet: • $T_{\text{normal\_duration}} = T_{\text{base\_unit}} \times T_{\text{n\_normal}}$ • $T_{\text{actual\_duration}} = T_{\text{base\_unit}} \times T_{\text{n\_actual}}$
`TupletEnd`	End of a tuplet	• T_{voice}: Voice index
`EndOfScore`	End of the score	None

Table 1. Token types and their attributes

element. We define a set of rules that scores must follow to be considered valid. They are checked against a linearization of the score into a sequence of *tokens*, representing different musical elements. This sequence is parsed by a state machine that maintains a state about the partially analyzed token sequence, and checks the validity of each new token according to our ruleset. We first detail how we tokenize the score, then the implementation of our state machine, and finally the rules that define allowed transitions.

3.2.1 Tokenization of a Score

We define several token types with attributes that represent various musical elements, *e.g*. notes, rests, tuplets, bars, etc. In this work, we focus on encoding elements relative to the rhythmic organization of the score. However, note that this tokenization is generic and could be extended to include other elements such as key signature and clef changes, *etc*. All token types and their attributes are detailed in Table 1.

The score is tokenized sequentially, going through musical symbols in a chronological order. To avoid issues with the boundaries of repeated sections (with tied notes for example), we unfold all repetitions before tokenizing the score. The order of simultaneous notes and rests does not have any impact on the ruleset we use. Simultaneous notes in the same voice are wrapped in between **`ChordStart`** and **`ChordEnd`** tokens. Similarly, in-tuplet notes and rests are wrapped in between **`TupletStart`** and **`TupletEnd`** tokens. An example of such tokenization is given in Fig. 3.

3.2.2 State Machine

The next step consists in validating or invalidating a sequence of tokens according to a set of rules. For this purpose, we use a state machine that tracks the state of the partially parsed sequence. We call this state the *score state*, notated S, containing two attributes:

- S_{voices} – List of *voice states*, describing the current completion of each voice.
- S_{tied} – Set containing all currently tied notes.

We note S_0 a special state representing the initial state of the machine, which will only be used later on to ensure that a score starts with a **`Bar`** token.

A voice state v has the following attributes:

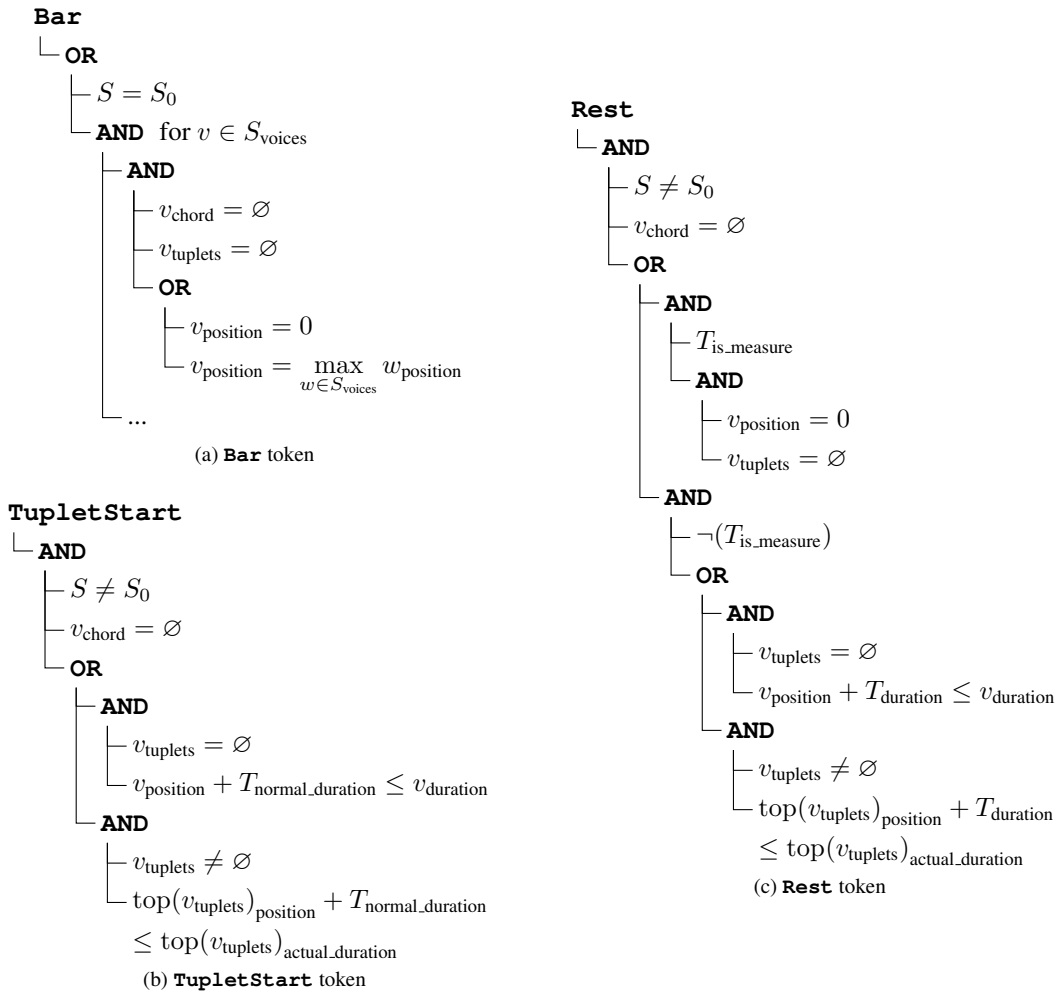
- v_{duration} – Duration of the voice, in quarter notes, derived from the current time signature.
- v_{position} – Current position in the voice.
- v_{chord} – Either a *chord state*, when inside a chord (*i.e*. after a **`ChordStart`** token), or $\varnothing$ otherwise.
- v_{tuplets} – Stack of *tuplet states*, representing potential tuplets in construction, with the most nested one at the top of the stack. It is $\varnothing$ when there are no ongoing tuplets. The function top retrieves the top element of the stack, *i.e*. the most nested tuplet.

A chord state C has the following attributes:

- C_{duration} – Symbolic duration of the chord.
- C_{notes} – List of notes in the chord.

A tuplet state K has the following attributes:

- K_{position} – Current position in the tuplet (relative to the tuplet inner metric).
- $K_{\text{unit\_duration}}$ – Base unit duration of the tuplet, *e.g*. a eighth note in a 3:2 tuplet of eighth notes.
- $K_{\text{n\_normal}}$ – Number of units that the tuplet should normally take. In the example above it would be 2.



(a) **Bar** token

(b) **TupletStart** token

(c) **Rest** token

Figure 4. Guard condition examples for a token T for a state S, notating $v = S_{\text{voices}}[T_{\text{voice}}]$

- $K_{\text{n\_actual}}$ – Number of units that the tuplet actually contains. In the example above it would be 3.
- $K_{\text{normal\_duration}} = K_{\text{n\_normal}} \times K_{\text{unit\_duration}}$
- $K_{\text{actual\_duration}} = K_{\text{n\_actual}} \times K_{\text{unit\_duration}}$

3.2.3 Guards of Transitions

Starting from the initial state S_0, we read the token sequence incrementally. For each new token, we ensure that the transition is valid before updating the state, by checking a verification condition called a *guard*. Each guard is token-specific and ensures that the current state can be updated by the new token through a valid transition. We detail some guards in Fig. 4, while an exhaustive list is available in Appendix B. If a guard condition is false at some point when parsing a sequence, this means that the score does not respect one of our rules, which raises an error.

For example, a **Bar** token (Fig. 4a) is allowed either at the very beginning of the sequence ($S = S_0$), or if all voices are complete, *i.e.* all chords have ended, all tuplets have ended, and all voices are synchronized [5] or empty.

A **Rest** token (Fig. 4c) is *not* allowed to come first in the sequence ($S \neq S_0$) and it cannot occur when a chord is in construction in the voice. To be valid, it must additionally be either a full-measure rest alone in the voice, or a regular rest which does not exceed the remaining duration of the voice – or tuplet if there is an ongoing tuplet in the voice.

Similarly, a **TupletStart** token (Fig. 4b) is *not* allowed as the first token of the sequence and cannot occur if there is a chord in construction in the voice. In addition, the tuplet's normal duration must not exceed the remaining duration of the voice – or outer tuplet if it is a nested tuplet.

Those guards could be extended to enforce stricter rules, or made optional to be more lenient with edge cases.

3.2.4 State Update

If the guard passes, then we can update the state of the machine. The current state is modified based on the new token and its attributes. These updates are generally straightforward. For example, **Bar** resets all voice positions to 0 and sets all voice durations according to its time signature. **Rest** advances the position of its voice by its duration. **TupletStart** pushes a new tuplet state onto the tuplets stack of the voice, that will be popped out when a **TupletEnd** occurs on that same voice.

[5] We recall that we do not consider incomplete voices as they have been filled with rests to fill the gaps.

4. IMPLEMENTATION AND APPLICATION

We assess the effectiveness of our error detection system by applying our syntactic checks on the MusicXML piano scores of the ASAP dataset [1].

4.1 Implementation Details

Our implementation is publicly available [6] .

4.1.1 Individual Errors

For individual errors, we parse the MusicXML file and compare the different durations as explained in Section 3.1.

4.1.2 Tokenization

We first parse the MusicXML files using Partitura [16], which keep the timeline untouched, meaning that overlapping notes in MusicXML still overlap in Partitura's internal representation. We then iterate chronologically over all elements on the timeline, creating tokens for supported ones.

Using Partitura's representation as input for our tokenizer allows the contextual error detection method to support not only MusicXML, but also all other score formats supported by Partitura. It could even be extended to unsupported formats by implementing the tokenization logic for those.

4.1.3 State Machine

We implement the states and guarded transitions introduced in Section 3.2. While nested tuplets are allowed in our framework, we chose not to implement them as they should be extremely rare, and the ones in the ASAP dataset are mainly due to encoding errors anyway. We also dropped support for grace chords, as Partitura currently does not support them. Instead, they are parsed as sequential grace notes. This has no practical impact on our method.

4.2 Results on ASAP Dataset

ASAP [1] contains 235 MusicXML scores, aligned with performances. However, those scores were not written by professionals, and therefore include numerous mistakes and notational errors. We report a summary of the proportion of the dataset impacted by both types of errors in Table 2.

4.2.1 Check of Individual Errors

Among the 235 scores in ASAP, 65 are flagged for inconsistency between theoretical and actual duration of notes, representing around 28% of the dataset. Those errors are present in 692 measures, or 1.9% of the total number of measures in the dataset. Most erroneous scores contain only a few invalid measures, with the median number of bad measures per score being 3. Only 17 scores have more than 5% of measures containing this type of errors, and 5 more than 30%.

Figures 5 and 6 show some representative errors detected with this method. Most of them are either rounding errors due to a bad division value in the MusicXML, or invisible notes with unusual durations included solely for MIDI playback.

[6] https://github.com/leleogere/detecting-notational-errors-in-digital-music-scores

```
<measure>
  <attributes>
    <divisions>480</divisions>
    ...
  </attributes>
  ...
  <note>
    <duration>34</duration>
    <type>32nd</type>
    <time-modification>
      <actual-notes>14</actual-notes>
      <normal-notes>8</normal-notes>
    </time-modification>
    ...
  </note>
</measure>
```

Figure 5. Example of tuplet rounding error in measure 94 of Beethoven's Sonata No. 17, op. 32, 2nd movement. The theoretical duration of a 32nd note in a 14:8 tuplet should be $\frac{1}{8} \times \frac{8}{14} \times 480 = \frac{240}{7} \times 480 \approx 34.3 \neq 34$.

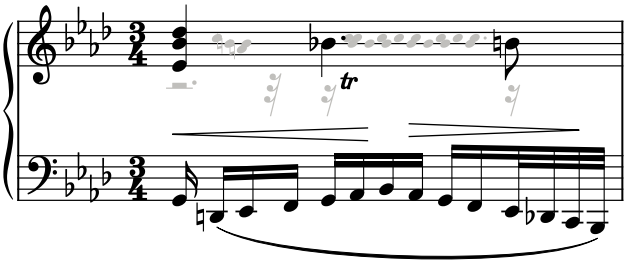


Figure 6. Example of invisible notes (in gray) of arbitrary duration resulting in a non-decomposable `<backup>` tag in measure 55 of the 1st movement of Beethoven's Sonata No. 31. Those notes were likely added for MIDI playback of the trill, and mess up with the MusicXML timeline.

4.2.2 Check of Contextual Errors

Because our state machine only deals with theoretical durations, and not with MusicXML timeline durations, individual errors will trigger contextual errors too as voices will be inconsistently filled. Therefore, we only run the contextual checks on the 170 correct scores left after the first stage, resulting in 35 incorrect scores:

- One had two parts, which we did not implement as a piano score should only have one part (containing multiple staves, but still one part).
- Two were only partially parsed by Partitura, which had trouble detecting some tuplet durations, which our tokenization procedure relies on. Those parsing difficulties likely point towards a notational error.
- The 32 remaining scores had been flagged as invalid by our set of rules.

Exploring those invalid scores, we successfully detected the following errors:

- **Measure overflow:** a voice is too long for the current time signature. This is generally caused by invisible rests (Fig. 7) or grace notes encoded as cue notes, which then have a non-zero duration (Fig. 8).
- **Same-voice notes overlap:** a note starts before the end of a former note in the same voice. While not

	Scores w/ errors	% scores w/ errors	Bars w/ errors	% of bars w/ errors	Median bars w/ errors in flagged score
Individual errors	65	$\frac{65}{235} = 28\%$	692	$\frac{692}{36496} = 1.9\%$	3
Contextual errors	35	$\frac{35}{170} = 21\%$	165	$\frac{165}{24648^*} = 0.7\%$	2

Table 2. Repartition of errors in ASAP dataset. The * indicates that the number of measures has been computed with unfolded scores, excluding the three incorrectly parsed MusicXML files mentioned in Section 4.2.2.

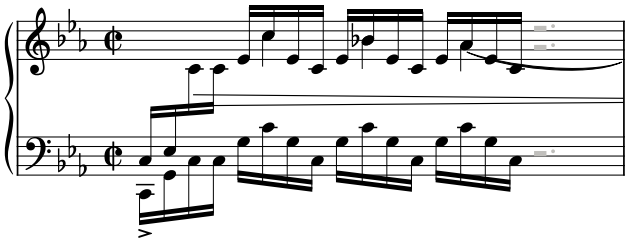


Figure 7. Measure overflow due to invisible rests (in gray) in measure 7 of Chopin's Etude Op. 25 No. 12.

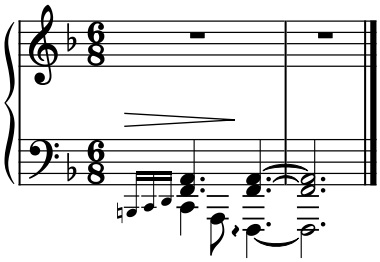


Figure 8. Measure overflow due to cue notes that should be grace notes, bar 97 of Liszt's Transcendental Etude No. 3.

directly detected as an overlap, this will trigger a measure overflow as our state machine does not have a direct notion of position; instead, it adds notes in the voice, eventually resulting in an overflow. For instance, in Fig. 2, the green voice is entirely filled by the first dotted quarter note, then the dotted eighth note coming next triggers an overflow as there is no more space in the measure.

- **Duplicate notes in chord:** as ASAP is a piano dataset, we forbid duplicated notes in a chord, as in Fig. 9.
- **Different durations in chord:** no such error were detected in ASAP. [7]

Note that the measure-overflowing mechanism also detects some occurrences of free played sections, such as in measures 232 to 234 of 1st movement of Beethoven's Piano Sonata No. 3 (Fig. 10). Those are considered as invalid by our model, as they go far beyond the nominal duration of the measure. These instances could be considered acceptable, and therefore, flagging them constitutes a false positive. This could be solved by allowing voices that are longer than the nominal measure duration, *i.e.*, making the overflowing rule optional. However, we argue this case (and other similar ones) is debatable: these small notes are

[7] In practice, our implementation does not find any such error as they are already fixed by Partitura's MusicXML import. Nonetheless, we confirmed that no such error exists in ASAP by directly parsing the MusicXML files.

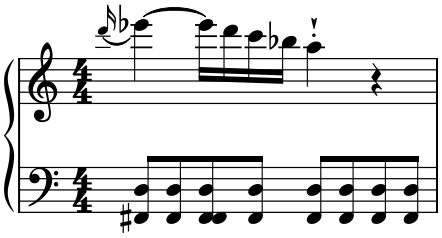


Figure 9. Duplicated note in chord in measure 97 of the 1st movement of Beethoven's Piano Sonata No. 21.

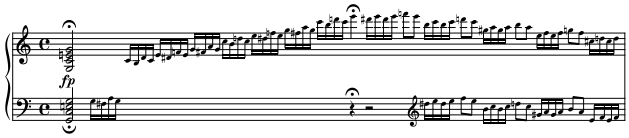


Figure 10. Free-played section in measure 232 of the 1st movement of Beethoven's Sonata No. 3. The actual duration of the measure is far longer than its nominal duration.

encoded as cue notes, which are supposed to be unplayed notes solely present to inform the performer about some other instrument's melody, not a free-play indication.

4.2.3 Fixing Scores

After detecting scores with notational errors, we have manually fixed some and pushed them to the ASAP repository. [8] Two of them contained tuplet rounding errors, which were fixed by computing an adequate value for the `<divisions>` tag, permitting an exact representation of all durations present in the score. All `<duration>` tags were updated according to this new value. Two scores were missing tuplet start tags, which would cause measure overflow issues later on. They were fixed by adding them back. Two scores were using incorrect tuplets, resulting in time-shifting and measure overflowing. One score contained extra invisible rests in an already-full measure, so we removed them. Finally, four other scores have also been fixed for other unrelated issues (extra staves, notes wrongly marked as invisible, poor notation choices).

5. DISCUSSION

In this article, we introduce a two-step method for the detection of notation error in digital music scores. The first step, specifically tailored for MusicXML, consists of ensuring consistency between different redundant duration information in MusicXML elements, in order to detect individual

[8] `https://github.com/fosfrancesco/asap-dataset`, only available on the `develop` branch as of October 2025.

note errors. The second step, format-agnostic, relies on an implemented tokenizer which linearizes the score into a sequence of tokens. This sequence is then checked by a state machine, whose role is to test the validity of each token within the musical surrounding context, in order to detect contextual errors.

Combining those two steps, our method is able to detect errors in 100 of the 235 scores of the ASAP dataset, or around 42%, confirming the fact that these scores were not written by professionals, and highlighting the interest of such detection methods. This allowed us to fix multiple scores in the dataset, enhancing its global quality. Scores that were manually fixed have been backported to the ASAP repository, for others to benefit from our work, even though there is still a long way to go to get a clean dataset.

Our method and implementation are modular and could be easily extended with more rules encoding various score quality aspects [15], to identify other types of errors. For instance, we might want to ensure that a score respects an specific instrument's tessitura, or make sure that no tie crosses a key signature or clef change – while not strictly forbidden, it is confusing to read. Hence, this model might serve as a base to build a more complete score validator to assess the quality of both human-produced and computer-generated music scores.

The main limitation of our approach is that while we detect errors, we do not automatically fix them. It is possible to suggest fixes for some error types, such as proposing a more suitable `<duration>` tag to correct duration inconsistencies. However, this still requires manual intervention and some expertise to check that the fix is suitable. Indeed, to fix rhythm/time consistency errors on notes and rests, one would need to choose which information becomes authoritative: the theoretical duration computed from the symbol or the timeline duration. The `<forward>`/`<backup>` tags, introduced to keep up with alignment in case of rounding errors, are even more difficult to deal with. Conversions between software can introduce invisible rests that are difficult to clean up automatically. Contextual errors are especially hard to automatically fix, because they require understanding the intention of the composer to sort out the cause of the error (*e.g.*, why is a measure overflowing?) and some expertise to know how to correct it while following the conventions of music notation. We observed that mistakes often tend to be very specific to one score, as two individuals have different ways of transcribing a piece. While a fully automated correction pipeline might be out of reach, our work remains useful to flag issues in scores. It then remains up to the user to inspect the problematic measures and determine the most appropriate fixes, *e.g.*, by comparing to references of that score in other published editions if they exist.

6. REFERENCES

[1] F. Foscarin, A. McLeod, P. Rigaux, F. Jacquemard, and M. Sakai, "ASAP: a dataset of aligned scores and performances for piano transcription," in *International Society for Music Information Retrieval Conference (ISMIR)*, 2020, pp. 534–541.

[2] "Basic Notated Music | Humdrum." [Online]. Available: https://www.humdrum.org/rep/kern/

[3] "abc | home." [Online]. Available: https://abcnotation.com/

[4] M. Good, "MusicXML for Notation and Analysis," in *The Virtual Score, Volume 12: Representation, Retrieval, Restoration*. The MIT Press, 2001.

[5] "MusicXML for Exchanging Digital Sheet Music." [Online]. Available: https://www.musicxml.com/

[6] P. Roland, "The music encoding initiative (MEI)," in *Proceedings of the First International Conference on Musical Applications Using XML*, vol. 1060. Citeseer, 2002, pp. 55–59.

[7] "Music Encoding Initiative." [Online]. Available: https://music-encoding.org/

[8] G. Read, *Music Notation: A Manual of Modern Practice*, 2nd ed. Crescendo Publishers, 1979.

[9] E. Gould, *Behind Bars: The Definitive Guide to Music Notation*. Faber Music Ltd, 2011.

[10] W. Brodsky, Y. Kessler, B.-S. Rubinstein, J. Ginsborg, and A. Henik, "The mental representation of music notation: notational audiation," *Journal of Experimental Psychology: Human Perception and Performance*, vol. 34, no. 2, pp. 427–445, 2008.

[11] N. Nápoles, G. Vigliensoni, and I. Fujinaga, "Encoding matters," in *Proceedings of the 5th International Conference on Digital Libraries for Musicology (DLfM)*. Association for Computing Machinery, 2018, p. 69–73.

[12] S. Mohammed, L. Budach, M. Feuerpfeil, N. Ihde, A. Nathansen, N. Noack, H. Patzlaff, F. Naumann, and H. Harmouch, "The effects of data quality on machine learning performance on tabular data," *Information Systems*, vol. 132, p. 102549, 2025.

[13] D. Fiala, P. Rigaux, A. Tacaille, V. Thion, and G. Members, "Data Quality Rules for Digital Score Libraries," IRISA, Université de Rennes, Research Report, 2018.

[14] F. Foscarin, D. Fiala, F. Jacquemard, P. Rigaux, and V. Thion, "Gioqoso, an online Quality Assessment Tool for Music Notation," in *Proceedings of the International Conference on Technologies for Music Notation and Representation (TENOR)*. Concordia University, 2018.

[15] F. Foscarin, P. Rigaux, and V. Thion, "Data Quality Assessment in Digital Score Libraries. the Gioqoso Project," *International Journal on Digital Libraries*, vol. 22, no. 2, pp. 159–173, 2021.

[16] C. Cancino-Chacón, S. Peter, E. Karystinaios, F. Foscarin, M. Grachten, and G. Widmer, "Partitura: A Python Package for Symbolic Music," in *Proceedings of the Music Encoding Conference (MEC)*, 2022.

A. EXAMPLE OF A MUSICXML CONTAINING NOTATIONAL ERRORS

The XML in Fig. 11 is the source of the examples from Fig. 1. There are some duration inconsistencies in notes, as well as unusual <backup>/<forward> tags crossing the measure boundaries. Different pieces of software deal with those differently, resulting in rendering differences. This illustrates how the MusicXML specification does not enforce strict respect of musical notation, and allows for ambiguous interpretation when engraving.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE score-partwise PUBLIC "-//Recordare//DTD MusicXML 4.0 Partwise//EN"
↪  "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise version="4.0">
  <part-list>
    <score-part id="P1">
      <part-name>Piano</part-name>
      </score-part>
    </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <!-- In this file, a quarter note has duration 2 -->
        <divisions>2</divisions>
        <time>
          <beats>2</beats>
          <beat-type>4</beat-type>
          </time>
        <clef>
          <sign>G</sign>
          <line>2</line>
          </clef>
        </attributes>
      <note>
        <pitch><step>C</step><octave>4</octave></pitch>
        <voice>1</voice>
        <duration>4</duration>
        <type>half</type>
        </note>
      </measure>
    <measure number="2">
      <note>
        <pitch><step>E</step><octave>4</octave></pitch>
        <voice>1</voice>
        <!-- /!\ Half note with a duration of 2 instead of 4 -->
        <duration>2</duration>
        <type>half</type>
        </note>
      <note>
        <pitch><step>G</step><octave>4</octave></pitch>
        <voice>1</voice>
        <!-- /!\ Same as above -->
        <duration>2</duration>
        <type>half</type>
        </note>
      <backup>
        <!-- /!\ How to deal with this backup of duration 6 when we are only 4 divisions into the
        ↪  measure? -->
        <duration>6</duration>
        </backup>
      <note>
        <pitch><step>B</step><octave>4</octave></pitch>
        <voice>2</voice>
        <duration>2</duration>
        <type>quarter</type>
        </note>
      <backup>
        <!-- /!\ again a backup with a weird duration -->
        <duration>6</duration>
        </backup>
      <note>
        <pitch><step>D</step><octave>5</octave></pitch>
        <voice>3</voice>
        <duration>2</duration>
        <type>quarter</type>
        </note>
      </measure>
    </part>
  </score-partwise>
```

Figure 11. XML source of Fig. 1.

B. GUARD CONDITIONS

We detail below the guard conditions for the token types that are not introduced in Section 3.2. In each figure, we test whether the newly read token T allows for a valid transition while the state machine is in the state S.

`Note`

- **AND**
  - $S \neq S_0$
  - **OR**
    - **AND**
      - $v_{\text{chord}} \neq \varnothing$
      - $T_{\text{duration}} = (v_{\text{chord}})_{\text{duration}}$
      - $T_{\text{is\_grace}} = (v_{\text{chord}})_{\text{is\_grace}}$
    - **AND**
      - $v_{\text{chord}} = \varnothing$
      - **OR**
        - **AND**
          - $T_{\text{is\_grace}}$
          - $v_{\text{position}} < v_{\text{duration}}$
        - **AND**
          - $\neg\,(T_{\text{is\_grace}})$
          - **OR**
            - **AND**
              - $v_{\text{tuplets}} = \varnothing$
              - $v_{\text{position}} + T_{\text{duration}} \leq v_{\text{duration}}$
            - **AND**
              - $v_{\text{tuplets}} \neq \varnothing$
              - $\text{top}(v_{\text{tuplets}})_{\text{position}} + T_{\text{duration}} \leq \text{top}(v_{\text{tuplets}})_{\text{actual\_duration}}$

Figure 12. Guard condition for a **`Note`** token

`EndOfScore`

- **OR**
  - $S = S_0$
  - **AND**
    - **AND** for v in S_{voices}
      - **AND**
        - $v_{\text{chord}} = \varnothing$
        - $v_{\text{tuplets}} = \varnothing$
        - **OR**
          - $v_{\text{position}} = 0$
          - $v_{\text{position}} = \max_{w \in S_{\text{voices}}} w_{\text{position}}$
      - ...
    - $S_{\text{tied}} = \varnothing$

Figure 13. Guard condition for an **`EndOfScore`** token

`ChordStart`

- **AND**
  - $S \neq S_0$
  - $v_{\text{chord}} = \varnothing$
  - **OR**
    - **AND**
      - $T_{\text{is\_grace}}$
      - $v_{\text{position}} < v_{\text{duration}}$
    - **AND**
      - $\neg\,(T_{\text{is\_grace}})$
      - **OR**
        - **AND**
          - $v_{\text{tuplets}} = \varnothing$
          - $v_{\text{position}} + T_{\text{duration}} \leq v_{\text{duration}}$
        - **AND**
          - $v_{\text{tuplets}} \neq \varnothing$
          - $\text{top}(v_{\text{tuplets}})_{\text{position}} + T_{\text{duration}} \leq \text{top}(v_{\text{tuplets}})_{\text{actual\_duration}}$

Figure 14. Guard condition for a **`ChordStart`** token

`ChordEnd`

- **AND**
  - $v_{\text{chord}} \neq \varnothing$
  - $\left|(v_{\text{chord}})_{\text{num\_notes}}\right| \geq 2$

Figure 15. Guard condition for a **`ChordEnd`** token

`TupletEnd`

- **AND**
  - $v_{\text{tuplet}} \neq \varnothing$
  - $\text{top}(v_{\text{tuplets}})_{\text{position}} = \text{top}(v_{\text{tuplets}})_{\text{duration}}$

Figure 16. Guard condition for a **`TupletEnd`** token