`tScore`: Making Computers and Humans Talk About Time

Markus Lepper[1] and Baltasar Trancón y Widemann[1,2]
[1] `<semantics/>` *GmbH*
[2] *Ilmenau University of Technology*
post@markuslepper.eu, baltasar.trancon@tu-ilmenau.de

Keywords: Knowledge Acquisition, Temporal Data, Man–Machine Interface, Domain-Specific Languages

Abstract: Textual denotation of temporal data is a challenge. In different domains very different notation systems have been developed for pen and paper during history, reflecting domain specific theory and practice. They work with symbolic representation of activities and combined expressional and spatial representation of time. In contrast, existing computer-readable formats are either simple lists or complicated expression languages. The formalism and software presented here, called `tScore`, try to bridge the gap between these two worlds by studying the most advanced example of the former, the Conventional Western Notation of music. By abstracting its basic principles, a generic notation framework is defined, suited for reading and writing by both humans and computers. The syntactic framework of the front-end representation and a mathematical formulation of the underlying semantics are given, which both are parametrisable and allow to plug in application specific parsers and data models. The current state of library implementation is shortly sketched, together with a practical example of moderately complex music notation.

1 Motivation and Design Goals of `tScore`

In the first place, `tScore` is the name of a new *text format* for denotating *arbitrary time-related* information structures. Furthermore, it is the software framework for processing this information.

The development of `tScore` starts from the observation that a temporal description language is highly desirable which ...

- can be read and written equally well by humans and computers,
- is neither restricted to one fixed model of "time" nor to prescribed parameter values,
- and can be handled with paper and pencil, or with chalk and blackboard, as the minimally required information processing hardware.

There are many potential application areas for such a language in the fields of handling technical systems and of æsthetic production. In general it enables the communication between users and computer systems for the purposes of

- automated performance (sequencing),
- automated transformation/generation of temporal data,

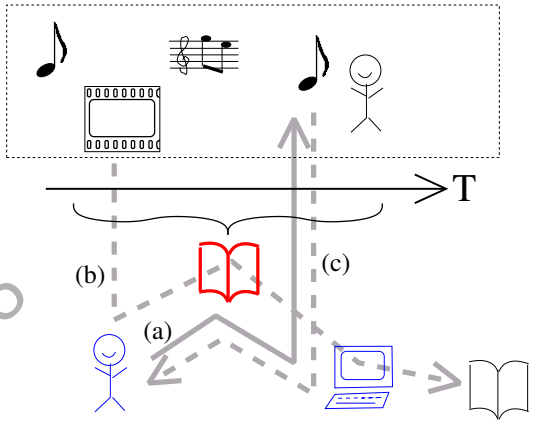

Figure 1: Information Flow Between User, Computer and Stage

- computer aided analysis,
- documentation and type setting into multiple formats.

The most complex use cases arise in the context of æsthetic production and live performance. Figure 1 shows different flows of information related to some live stage activity between a human user and a computer system: (a) The user writes a score which is "sequenced" or "performed" by the IT system, (b) the

ISBN 978-989-8565-81-5

user writes a protocol of what he/she perceives, which is rendered nicely by the IT system, (c) the computer writes a protocol (using motion/gesture/pitch detection technologies) and delivers a readable score to the human. In all theses cases (and many more one can think of) a `tScore` score object can be the central means for information exchange. The motivation of the authors came out of a concrete composition project, where it is necessary to unify "Conventional Western Music Notation" (CWN) and technical control parameters for electronic sound processing in one single human- and computer-readable score.

But `tScore` as such shall not be restricted to these two domains. In fact, it is one of its central design criteria that the "time axis" as well as the "value range" are *in no way a priori restricted*, but can be mapped to almost arbitrary user-defined domains.

The main problem w.r.t. the time axis is that even the basic *notion* of time, the assumptions about its behaviour and structure, the permitted denotations and algebraic operations, the whole so-called *model of time*, shall not be defined in advance, but definable by the user, fitting the application context. The value axis is in most cases less critical, but, e.g. in case of CWN, subject of controversial discussions and cannot be pre-defined either.

Of course, all these necessary definitions need not be reiterated from scratch, but the user is given a collection of pre-defined library components, which can be parameterised and plugged together.

So `tScore` is intended to serve as a denotation of temporal structures in very diverse contexts, like

- light control
- video cue lists
- kinetic sculptures / robots
- dramaturgy / radio play
- web animation
- stage performance
- music
- *(every other conceivable time-related structure)*

2 Existing Approaches to Denotate Time

The existing approaches for denotating temporal structures can be arranged on a scale from "analog" to "symbolic", following the categories of GOODMAN (Goodman, 1976):

- Modern variants of pure graphical music notation are "analog" encodings. This means that distances in time are proportionally represented by distances on the writing surface, in one distinguished direction.

  The traditional German word is "Streckennotation", and the technique is employed in avantgarde music, mixed with classical notation elements or even exclusively. The different value aspects which change in time are denotated either by traditional pitch indication, using five line musical note systems, or by some "y-axis" which interpretes some curved lines.

- On the opposite extreme are the "symbolic" representations. In the field of music these are the pure expression languages, like "musixTEX" (Taupin et al., 2002), "Guido", "lilypond" (lilypond, 2011), "musicXML" (musicxml, 2011) and many more. Here no relation between spatial location and time is defined. Only the mere sequential order of sub-terms carries any semantics, in many cases denotating the mere sequential order in time. Duration, synchronicity, temporal distances and parallelism are expressed by expression terms only, built from lexical atoms and combining operators.

- Somewhere in the middle, more to the symbolic side, is the classical music notation CWN. It is a term language because the *exact* values, e.g. of durations, are encoded by the chosen lexical atoms and combining operators (e.g. beams). Nevertheless, a certain flavour of analog representation is also present: The distance of the "terms", the note symbols, and their horizontal alignment across different, simultaneously played voices, is expected to "redundantly" represent the temporal positioning as spatial layout. The balance of analog and symbolic aspects has been carefully analysed by DAHLHAUS (Dahlhaus, 1965).

But all of these existing approaches lack the desirable flexibility. The above-mentioned problem how to integrate CWN and technical control parameters is in practice usually solved by "abusing" some formalism, e.g. by a brute-force mapping of "Midi" control values to some semantic user domain without any type check or declaration.

This ubiquitous practice is a "hack" and neither adequate to professional software engineering nor to informatics as the scientific discipline which analyses culturally determined information structures.

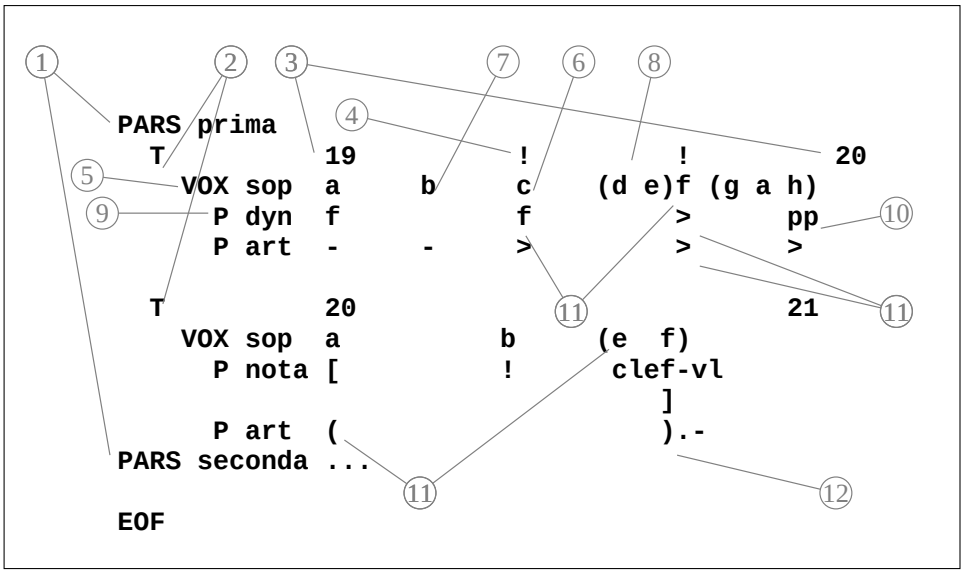

Figure 2: Example of the `tScore` Input Format

3 `tScore` As a Consequence From CWN

The CWN has been developed over hundreds of years, and is the result of both research results of dedicated specialists, and of every-day practical experience of the musical community. Consequently, it has a lot of advantages:

- It is highly ergonomic, economic and compact, due to the requirements and experiences of musical practice.
- It is flexible, parametrisable and adaptable: With some "plugged-in" additional definitions it can cover a large historic era from PEROTIN to STOCKHAUSEN.
- As mentioned above, it can be "abused" for notating temporal structures of value ranges outside the traditional realm of music, e.g. physical motion or film rhythms.

Naturally, the principle of historic development led to detours and idiosyncrasies:

- There are multitudes of notational elements for expressing the same thing, e.g. duration by note heads, stems, flags, dots, ties and tuplet brackets, which interact in a complicated way.
- The (mathematically spoken) *domain* and the *ranges* of the denotated (mathematical) function are restricted to metrical time and to traditional pitch information.
- But even when applying this restriction, there is a multitude of semantic interpretations, with partly contradictory assumptions.
- It is hardly readable by computers.

The design of `tScore` is the consequence of generalizing the "orthogonal" aspects of CWN:

- Time flows from left to right, top to bottom where lines must be broken.
- Temporal distribution is indicated by dividing space.
- Synchronicity is established by super-position and horizontal alignment.
- Multiple voices run in parallel, which again consist of multiple parameter tracks.

On the other hand `tScore` avoids the above-mentioned restrictions and deficiencies of CWN and ad-hoc computer languages:

- There is *no pre-defined* model of time, but only minimally necessary basic definitions.
- There are *no pre-definitions* on the value axes, but the user has to "plug" the required syntactic and semantic structures into the abstract framework.
- It allows arbitrary lexical identifiers for value denotation, and arbitrary *overloading*, organized by parameter tracks.
- The front-end representation of `tScore` is mere *type-writer text*. This means a simple linear sequence of characters. ASCII suffices, and a larger subset of Unicode is possible, as long as a monospace font is used to establish the y-coordinate as *text column*.

4 Syntax of the `tScore` Input Format

Figure 2 shows the basic elements of the `tScore` input format:

1. The reserved keyword "**PARS**", followed by a unique identifier, separates independent parts of a score file.
2. The reserved keyword "**T**" marks the beginning of a dedicated line of input text called *time line*.
3. Textual entries (of arbitrary format) in such a time line link the text column of their appearance to some instant of the chosen time domain.
4. The horizontal ranges between two neighbours of this time instances can be sub-divided by exclamation marks. (The resulting ranges can further be divided by dots, which is not shown in this example.)
5. The reserved keyword "**VOX**", followed by an identifier, marks the beginning of a so-called "voice-line", which is used to construct a single *voice* object. A voice is defined as a sequence of adjacent, non-overlapping *events*. Each such event is thus related to one particular voice and one particular time instant as its *coordinates*.

   One single lexical entity appearing in a voice line has two-fold semantics: Firstly it declares the mere *existence* of one single event with the coordinates corresponding to its position in the text, namely line and column number.

   Secondly it defines the value of one of the events *parameters*, namely the chosen *main parameter*.[1]
6. Such an event-defining entity can appear at a text column which is already defined by a temporal mark in the time line, ...
7. ...or it can appear in between, which defines a further level of ("spontaneous") sub-division of the temporal interval.
8. In a voice line, arbitrarily nested round parentheses can be used for further sub-division: The "logical time" is distributed evenly between all single events and parenthesized expressions appearing in one time segment. The same rule holds for the contents of the latter, and hence recursively for arbitrarily nestings.
9. The reserved keyword "**P**", followed by an identifier, starts a line which defines a further parameter for the events of the current voice.
10. Parameter values follow some arbitrary syntax, defined with the name of the parameter track. They can, but need not, appear in every column which is bound to an event. (But cannot define new events and time instances of their own.)
11. Esp. *overloading* is easily possible, the lack of which is a severe deficiency of nearly all competing systems. Here "`f`" is used for a pitch and for an intensity, "`>`" for a diminuendo fork and an articulation mark, and "`(`" as grouping in the event defining line, as described above, and as sign for *legato* in the parameter line.
12. The lexers for the different parameter tracks may support arbitrarily defined "ascii art".

[1] Two remarks:
First: The divisions of the horizontal text areas (numeric constants dividing the whole time line, exclamation marks between the numbers, events between the exclamation marks, parentheses between events, etc.) contribute only with their *mere number*: A division by three(3) has the same semantics independent of the numbers of concrete text columns and appearing white space characters in between.

Second: the very last column of a time line may *not* carry any event; any event starting at this time point must be notated at the beginning of the corresponding system, which is normally the next following in the score file.

5 First Parsing Phase and Generic Semantics

The first parsing steps convert the two-dimensional input, as described in the preceding section, into an intermediate data model. In the technical sense, in the sense of classical compiler construction, this data model already plays the role of a "semantic model" . Seen from the application's viewpoint, it is *not* a semantic model, but a generic and intermediate one, and the real semantics will be constructed by subsequent transformations, defined by the user. Figure 3 depicts these both levels in an informal way.

More precise are the formulas in Table 1. Let S be the set of all trimmed string values, and S_1 its subset without the empty string, and *Id* the strings usable as alphanumeric identifiers. The central notions are those of an "event" E and a "voice" V.

The time points on this level of abstraction represent "mere syntactical time". They are constructed as the algebraic data type $T_{\texttt{synt}}$, with certain consistency conditions.

They start with the top-level time points, represented by the constructor `top()` applied to some text argument. These represent the columns in the source text which are marked by some textual entry (but not by the subdivision signs "`!`" or "`.`") in the "`T ...`" time ruler line, see number (3) in Figure 2. Please

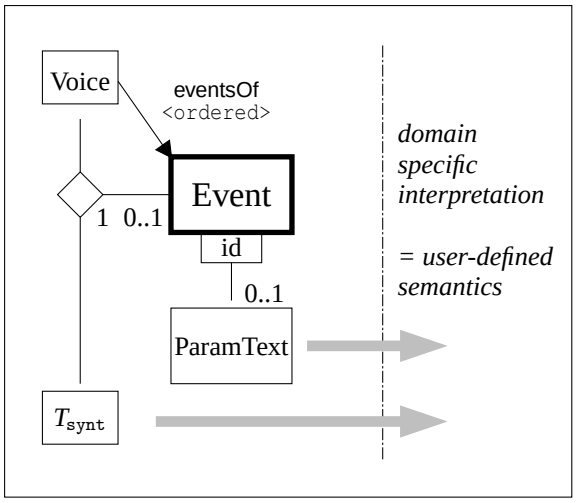

Figure 3: The `tScore` first step generic model, symbolic

note that the contents of these texts are totally left open. In many cases they will be numeric, representing measure numbers in case of CWN, or physical time units, etc., and ascending order will be required. But this must be defined in the subsequent user-defined transformation step, as indicated by the broad arrow symbol in Figure 3. On this level of definition, *arbitrary* text is allowed.

The further horizontal divisions induced by the source text (either on the time scale line, or in the event generating voice lines, see preceding section for details) are represented by the application of div, which takes two existing time points, divides the interval into *count* sub interval's, and takes the start of the *n*-th sub interval as a new time point, where *n* is encoded as *pos*. This data type is restricted, as *from* must be earlier than *to*, *count* must be greater or equal to two(2), and *pos* must be greater zero(0) and less than *count*. Again, there are no assumptions on particular methods for dividing intervals, i.e. there are no equalities like "$2/4 = 1/2$" imposed on this generic level.

Every event from E is related to one single voice from V and one single timepoint T_{synt}, and uniquely identified by their combination. The map eventAt is a (partial) bijection: At most one event occurs at a given time instance in a given voice. Conversely, every event is related to one voice and one time instance.

Beyond this role as mere labels for events and their sequential order, no further properties of T_{synt} are assumed. All *semantic* properties, and their significance for the events related to them, must be calculated in the subsequent, user-defined processing by mapping them to some domain specific model of time. This may impose equalities and further restrictions.

$$\mathsf{disjoint}(S,E,V)$$
$$S ::= \textit{all trimmed string values}$$
$$Id \subset S \qquad S_1 = S \setminus \{\varepsilon\}$$
$$\begin{aligned} T_{\text{synt}} ::= {} & \mathsf{top}(S_1) \\ \mid {} & \mathsf{div}(\textit{from}, \textit{to} : T_{\text{synt}}; \textit{count}, \textit{pos} : \mathbb{N}) \end{aligned}$$
$$\mathsf{eventAt} : (V \times T_{\text{synt}}) \rightarrowtail\!\!\!\!\!\!\rightarrow E$$
$$\mathsf{eventsOf} : V \to E^*$$
$$\mathsf{paramText} : (E \times Id) \rightarrowtail\!\!\!\!\!\!\rightarrow S$$

$$\mathcal{U} = \prod j : Id; e : E.\, (V_j \sqcup \{\bot_{\texttt{noData}}\})$$
$$L_j : S \to (S \times V_j) \cup \{\bot_{\texttt{noMatch}}\}$$
$$\oplus_j : (V_j \cup \{\bot_{\texttt{noData}}\}) \times V_j \to V_j$$
$$L_{j,e} : (S \times \mathcal{U}) \to (S \times \mathcal{U})$$
$$\frac{L_j(s) = (s', v) \qquad |s| > |s'|}{L_{j,e}(s,u) = (s', u \oplus \{(j,e) \mapsto (u(j,e) \oplus_j v)\})}$$
$$\frac{L_j(s) = \bot_{\texttt{noMatch}}}{L_{j,e}(s,u) = (s,u)}$$
$$\mathsf{ScoreFormat} : Id \rightarrowtail\!\!\!\!\!\!\rightarrow {L_j}^*$$
$$\mathsf{parse1param} : E \times Id \to \mathcal{U}$$
$$u_0 = Id \times E \times \{\bot_{\texttt{noData}}\}$$
$$\frac{\begin{array}{c}\mathsf{ScoreFormat}(i) = \langle c_{j_1}, \ldots, c_{j_n} \rangle \\ \mathsf{paramText}(e,i) = s \\ \lim\limits_{k \to \infty} (c_{j_1,e} \fatsemi \ldots \fatsemi c_{j_n,e})^k (s, u_0) = (\langle\rangle, u)\end{array}}{\mathsf{parse1param}(e,i) = u}$$
$$U = \bigcup e : E; i : Id.\, \mathsf{parse1param}(e,i)$$

Table 1: Data model of `tScore`. Upper part: pure syntactic data. Lower part: user-defined interpretation of parameter values.

Every combination of event and *Id* can point to at most one arbitrary string constant as the paramText. This is the text input as extracted from the source text, related to that event and appearing in the parameter track with the given id, see preceding section and Figure 2. Its meaning is, again, constructed in the subsequent user-defined transformation.

6 Second Parsing Phase and User-Defined Semantics

The basis to model all user-defined semantics is the unconditional (object-oriented, "co-algebraic") self-identity of the events from E. The finally resulting user data is an indexed collection U from Ta-

ISBN 978-989-8565-81-5

```
  T          19          !            !         20
    VOX sop  a     b     (c  d)  e    f    g
      P dyn  p<>1  !1     >1 !1  >1   !1ff0
// written in two lines is the same:
//    P dyn  p<                      ff
//           >1    !1     >1 !1  >1   !1
      P mat  ["Siegfriedmotiv"              ]
```

The resulting data structures are (events represented by their "pitch"):

$$\begin{aligned}\{\ & \mathsf{Group}(num = 0, start = "p <", end = "ff", events = \{a,b,c,d,e,f\}),\\ & \mathsf{Group}(num = 1, start = ">", end = "", events = \{a,b\}),\\ & \mathsf{Group}(num = 1, start = ">", end = "", events = \{c,d\}),\\ & \mathsf{Group}(num = 1, start = ">", end = "", events = \{e,f\}),\\ & \mathsf{Group}(num = 1, start = \text{"Siegfriedmotiv"}, end = "", events = \{a,b,c,d,e,f,g\})\\ \}\end{aligned}$$

Figure 4: Tendency and Groups Collector

ble 1, where V_j are the different specific ranges of user-level parameter values, indexed by some identifier $j \in Id$. The transformation from the values of paramText from the preceding section into these data is executed by a co-operation of two transformation steps. The basic idea is as follows:

First, there are *lexical parsers* L_j which (possibly) consume parts of the raw paramText string data from S and construct a value of some user-level parameter domain V_j. This domain must be a monoid together with the operation $\oplus_j$ and $\perp_{\texttt{noData}}$ as neutral element, which are used to combine the results of zero or more successful parsings. V_j may include a dedicated value resulting from conflicts and cases of error, which is treated as a normal parameter value in this stage of processing.

Second, by selecting one event $e \in E$ as current index, we construct a collector $L_{j,e}$ as a lexer with result storage $\mathcal{U}$. Seen as a function on $S \times \mathcal{U}$, it is decreasing in its first and increasing in its second component.

Finally, a ScoreFormat is a mapping from those explicit identifiers i which appear in the confirming scores as names of parameter tracks, like "`dyn`", "`art`" and "`nota`" in Figure 2, to a sequence of those parsers (with implicit identifiers j) which will be applied to the raw input parameter. (The voice line after the keyword "`VOX`" is treated the same way, by substituting the implicit parameter name `$main` $\in Id$.)

The parse result for one event $e \in E$ and one parameter id $i \in Id$ is realized by the function parse1param: The parsers are instantiated to collectors for e and applied in turn to the input until the result stabilizes. This is guaranteed to occur in finite time, because every proper change to u is accompanied by some consumption of s. When s is totally consumed then parsing is finished; when no shortening of s has taken place for a whole loop then an error condition is detected.

Following this framework, in case of success we get at last a complete data storage U which relates every event to an indexed collection of user-defined value types.

The translation of **time values** is much more specific and much less subject to automation: In most cases dedicated parameter values from U are involved to create the translation from $T_{\texttt{synt}}$ to some domain specific model of time and duration. In case of CWN these are meter indications, bar numbers, tempo indications, etc. We assume that in this field further attempts to automate will soon reach their limits, but versatile generic libraries must be provided anyhow.

7 Generic Building Blocks, Current Implementation

For constructing the implementation of the user-defined semantics, the current `tScore` implementation comes with a library of generic code objects which can be parametrised and plugged together by the user. Currently this has to be done by writing source code in the Java language. Future development will include a more easy-to-use configuration language for replacing the low-level Java programming, as far as possible. The building blocks are generic and parametrisable, and belong to one of two groups:

First, there are the lexical analyses and translators which parse the text fragments from the parameter tracks of the input text into Java collections indexed by Event objects. They correspond to the lexical parsers L_j and collectors $L_{j,e}$ from the preceding

ISBN 978-980-8565-81-5

```
T          19          !          !       !          20
  VOX sop  a  b  c  d  b  c  d  e  %       c  d  e  f

    P art  $- .- (  )- sim        >       sim
//yields
//  P art  -  .- (  )- -  .- (  )- >       -  .- (  )-

    P ls   $f p  p  sim           TERM    cont
//yields
//  P ls   f  p  p  f  p  p  f  p         p  f  p  p
```

Figure 5: Pattern Distributor

section.

Secondly there are transformation algorithms which apply higher-level transformations on complete sequences of events. Some of them come from techniques typical for CWN, but most are applicable to all kinds of data values. Some of them operate on the sequences of events after parsing and interpretation, i.e. on user-level data. But most of them operate like pre-processors on the unparsed input values of paramText.

The most significant tools from this group are:

Dotted Notation Expander As known from CWN, a some kind of "dotted notation" can be very convenient for denotating the very frequent duration sequences like $(1-(2^{-n});2^{-n})$ for $n \geq 2$.

In `tScore` this is realized by a transformation which extracts and deletes the dots from the lexical representation of an *arbitrary* main parameter, and adjusts the proportions of the temporal coordinates *a posteriori*.

Tendency and Groups Collector This tool collects group of events into composite data objects. Figure 4 shows some examples: All event objects enclosed by the corresponding parameter denotations are collected to one group, identified accordingly.

Pattern Distributor Realizes the "*simile*" construct from CWN, see Figure 5: A dedicated marker (defaults to "**`$`**") starts a pattern definition, the keyword "**`sim`**" starts its repetition, and any explicit value in this parameter track, or the keyword "**`TERM`**" ends it. "**`cont`**" resumes execution of the pattern at the phase position where it has stopped.

Of course all these keywords are configurable, and the operation is totally independent from the value range of the track to which it is applied.

Duration Distributor and Placeholder Eraser As mentioned above, the basic notion of events is a duration-less instant in time. In most case the distance to the subsequent event is meant as the event's "duration parameter". This is realized by the duration distributor, which calculates this distance and assigns it to the event as the value of that parameter. From now on this event may be freely moved around and does not need any reference to its successor or voice context any more! Note that the calculation of this time distance may be a complicated task, depending on the chosen time model.

For pauses and prolongations, pseudo-events may be inserted into the event sequence. These are removed by the placeholder eraser, for prolongations typically *before* and for pauses *after* the duration calculation has taken place.

Running Octave Collector Implements the running octave or interval minimization discipline, well known from musixTEX, lilypond and many other input formats. This input feature infers the octave register of the next note of a melody, if it is not given explicitly, by choosing the smallest melodic step from its predecessor, and is applied e.g. in Figure 6

Metric Distributor In case the time model is CWN-like, then there must be a dedicated (pseudo-)voice which describes metrics, incomplete bars, tempo, etc. This processor distributes the current metric information to all top-level time points, enabling the subsequent translation of textual input columns into bar-relative metric positions.

8 State of the Work, Future Work

As mentioned above, the theory and implementation of the combinator library for the second phase of parsing (translating the generic data into user-defined data) is still subject of research. The next major step in the `tScore` project will be the research on the algebraic structure of the combinators $\oplus_j$ (see Table 1), on their relations to the corresponding *syntactical* combinators of the parsers input side, and the definition of a

```
// fuga_a_3.cwn
PARS p1
T         1                   2                3           4
 VOX M    2/4
 VOX sop  (g'2 ,c) (% es d c) 'as as as (f d)  g g g (es c)
   P nota  cl-Vl

T        4             5                   6                    7
 VOX sop f f (f d g f) (es f es d)(c    c')(- b a g)(fis g a -)
 VOX alt               (c'2  g)   (% b a g)'es es   es (c a)
   P nota               cl-g2
```

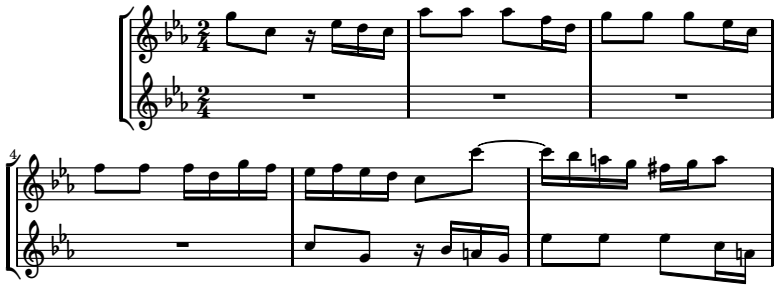

Figure 6: `tScore` CWN to Lilypond Example

library supporting both.

Beside the mathematical questions, in which standard methodologies from algebra and co-algebra will be applied, practical issues have to be addressed:

- Error handling must be supported in a user-friendly way, error recovery and diagnosis have to distribute over these combinators.
- The intended fully free design of front-end representations (see label number 12 in Figure 2 !-), must be supported by some a priori diagnosis for detecting possible ambiguities and for automated selection of adequate parsing techniques.
- A front-end language must be implemented to make the combinator library accessible without the need of genuine Java programming.

We are sure that the work invested in the notation of CWN will also turn out fruitful for information acquisition in totally different areas, where also domain specific "analog" short-hand notations for temporal structures are common to the domain experts.

Luckily the framework and our specialized libraries up to the current point work fine, and give a firm grid for further practical experiments and theoretical studies.

Our instantiation for CWN supports already some important kernel features beside mere note sequences, like metric change, incomplete measures and clef change. It converts into a semantic user model, and has a translation into "lilypond" note setting source text (lilypond, 2011)..

Figure 6 shows a `tScore` input file and the corresponding note print out created by this famous note setting program. You nearly can play the music *prima vista*, from *both* notations, can't you?

REFERENCES

Dahlhaus, C. (1965). Notation Neuer Musik. *Darmstaedter Beitraege zur Neuen Musik*, 9.

Goodman, N. (1976). *Languages of Art. An Approach to a Theory of Symbols.* Hackett Publishing.

lilypond (2011). *Lilypond Music Notation*. `http://lilypond.org`.

musicxml (2011). *MusicXML website*. `http://www.recordare.com/musicxml`.

Taupin, D., Mitchell, R., and Egler, A. (2002). MusixTEX — Using TEXto write polyphonic or instrumental music. `http://www.ctan.org/tex-archive/musixtex/taupin/musixdoc.pdf`.