Real-Time Matching of *Antescofo* Temporal Patterns

Jean-Louis Giavitto José Echeveste
IRCAM – UMR STMS 9912 CNRS & INRIA MuTant project & Sorbonne Universités - University of Paris 6
1 place Igor Stravinsky, 75004 Paris, France
[giavitto,echeveste]@ircam.fr

Abstract

This paper presents *Antescofo temporal patterns* (ATP) and their online matching. *Antescofo* is a real-time system for performance coordination between musicians and computer processes during live music performance. ATP are used to define complex events that correspond to a combination of perceived events in the musical environment as well as arbitrary logical and metrical temporal conditions. The real time recognition of such event is used to trigger arbitrary actions in the style of event-condition-action rules. The musical context, the rationales of temporal patterns and several illustrative examples are introduced to motivate the design of ATP. The semantics of ATP matching is defined to parallel the well-known notion of regular expression and Brzozowski's derivatives but extended to handle an infinite alphabet, arbitrary predicates, elapsing time and inhibitory conditions. This approach is compared to those developed in log auditing and for the runtime verification of real-time logics. ATP are implemented by translation into a core subset of the *Antescofo* domain specific language. This compilation has proven efficient enough to avoid the extension of the real-time runtime of the language and has been validated with composers in actual pieces.

Keywords timed regular expressions, event-driven programming, score following, timed and reactive system, domain specific language, computer music, *Antescofo*.

1. Introduction

Antescofo is a score following system that listens to a live music performance to track the position in the score and the tempo of a performer and to trigger accordingly electronic actions (computations).

In this paper, we extend the *Antescofo* real-time language dedicated to the specification of the electronic actions with *temporal patterns*. *Antescofo* temporal patterns (ATP) are a formalism for specifying sequences of discrete elementary events and time intervals fulfilling an arbitrary property, occurring one after the other, augmented with timing information and arbitrary logical conditions. *Antescofo* Temporal Patterns extend the idea of timed regular expressions [3] and are fitted to the expression of temporal conditions that appears in the writing of Interactive Music (also called Mixed Music *i.e.*, mixing in real-time the human performance and the electronic response). For instance, it is possible to specify a complex event E such as "*a repetition of the same note within* $3/2$ *pulses such that there is no occurrence of* E *in the previous* 5 *pulses*".

In fact ATP go strictly beyond propositional modal logic as one may express for example "*a repetition of a pitch* N *within* $f(N)$ *pulses, each lasting at least* $g(N)$ *pulses*" where f and g are arbitrary functions that return a number from a pitch. Nevertheless, checking that a prefix of a time-event sequence of inputs matches an ATP can be checked efficiently in real-time "without looking ahead". Preliminary validations in the context of real musical pieces show that the implementation in the Max/MSP environment [26] always reacted in less than 3 milliseconds (with a Max/MSP control rate of 2ms), which ensure the musical simultaneity needed.

In the rest of this section, we give some background on score following and motivate the use of temporal patterns to trigger actions beyond the strict scope of score following. Section 2 relates temporal patterns with other formalisms developed for instance in event processing, in online analysis of logs (for intrusion detection) and in the runtime verification for timed linear time temporal logic. Temporal patterns are introduced informally in Section 4 and a formal semantics is presented in Section 5. Temporal patterns are implemented in *Antescofo* by source-to-source translation into a core subset of the language using delays, nested conditionals and synchronous control structures. The first uses of temporal patterns in new musical pieces provide evidences that the performance of this implementation is sufficient for most of the applications. Conclusions in Section 7 summarize the work and sketch some perspectives.

1.1 Score Following

Human musicians have since long developed methods and formalisms for ensemble authoring and real-time coordination and synchronization of their actions. Bringing such capabilities to computers and providing them with the ability to take part in musical interactions with human musicians, poses interesting challenges for authoring of time and interaction and real-time coordination.

In this context, *automatic score following* has been an active line of research and development among composers and performers of Interactive Music for 30 years [11, 31]. An automatic score following system implements a real-time listening machine that launches necessary computer music actions in reaction to the recognition of events in a score from an incoming music signal.

We have proposed in [9] a novel architecture for score following, called *Antescofo*, where the artificial machine listening is strongly coupled with a domain-specific real-time programming language. The motivation is to provide the composers an expressive language for the *authorship* of interactive musical pieces and to provide the performers with an effective system to implement ef-

PPDP '14, September 08–10, 2014, Canterbury, United Kingdom.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2947-7/14/09…$15.00.
http://dx.doi.org/10.1145/2643135.2643158

ficiently *dynamic performance scenarios*. Using this dedicated language, the composer creates an *augmented score*. The augmented score includes the instrumental parts (*i.e.*, the specific events that should be recognized in real time), the electronic parts (*i.e.*, the electronic actions) and the instructions for their real-time coordination during a performance. Expressiveness is a primary concern and the language focuses on the writing of time as a semantic property rather than a performance metric [20]. During a performance, the machine listening in *Antescofo* is in charge of encoding the dynamics of the outside environment (*i.e.*, musicians) in terms of incoming events, tempo and other parameters from the polyphonic audio signal. The language runtime evaluates the augmented score and controls processes timed to unfold synchronously with the musical environment.

The *Antescofo* unique approach allows the specification of flexible and complex temporal organizations for compositional and performative purposes. It has been validated through numerous uses of the system in live electronic performances in contemporary music repertoire of composers such as Pierre Boulez, Philippe Manoury, Marco Stroppa... and adopted by various music ensembles such as Los Angeles Philharmonics, Berlin Philharmonics... to name a few.

1.2 Score Follower as Transducer

As explained above, an *Antescofo* augmented score is a specification of both an instrumental part and the accompaniment actions. The instrumental part is specified as a sequence of musical events such as *note*, *chords*, *trills*, *glissandi*... The sequence of events $e_1 e_2 \dots e_\ell$ is simply the translation in a textual format of the traditional graphical notation of the score to follow. An accompaniment action a_i is associated with the event e_i that trigger it.

Thus, from an abstract point of view, the reactive engine can be roughly seen as a linear finite state transducer (a Mealy machine) that waits for the notification of the occurrence of a notes to launch the associated actions, see Fig. 1. The notification of the occurrence of a note is done by the listening machine that is responsible to analyze the audio input and to detect the onset of a new note. A reaction of the reactive engine is a transition in the transducer whose underlying automaton models the score to follow.

This is a rough approximation because the reactive engine manages errors from the listening machine and from the musician (*e.g.*, a note can be missed), the score can include jumps (*e.g.*, to implement repeats like *da capo* specification), and most importantly, actions are not necessarily atomic and they unfold in time with complex synchronization and duration constraints. However, we can ignore these complications in a first stage and, without loss of generality, we can restrict our attention only to notes defined by a pitch.

$$\begin{tikzcd}
\boxed{s_0} \arrow[r, "\mathtt{G4}/a_{1,1}\ a_{1,2}"] & \boxed{s_1} \arrow[r, "\mathtt{B\#4}/"] & \boxed{s_2} \arrow[r, "\mathtt{G4}/a_3"] & \boxed{s_3}
\end{tikzcd}$$

Figure 1. Modeling the reactive engine as a transducer for the "augmented score" $\mathtt{G4}^{a_{1,1}\ a_{1,2}}$ B#4 $\mathtt{G4}^{a_3}$ (events are annotated with the actions they launch, written in superscript). In the automaton, a transition is labeled by the input events and the output actions. For the sake of the simplicity, we neglect the specification of duration in events.

1.3 Score Follower as Pattern Matcher

The previous standpoint is definitively *score oriented*, that is, actions are subordinated to the notes in the score. However, an alternative vision is possible by reversing the perspective: events become labels of some actions. In this approach, the reactive program is primarily organized through its actions, not through its events. A program is a set of rules $e \to a$ specifying for each action a, its triggering event e. The score is no longer modeled in the reactive engine which acts rather as a *pattern-matcher*, constantly seeking some event in the output stream of the listening machine to trigger the actions.

In this approach the pitches to recognize in the audio input can be seen as constant patterns and it is tempting to introduce pattern variables and logical guards, for example to specify easily that we want to trigger an action each time "a note is followed by the same note transposed by a fifth". Fig. 2 shows the occurrence of a pattern corresponding to a sequence of three consecutive notes x, y, z such that $x < y$ and $y > z > x$ (we refer here to the pitch of the notes). However we stress that the real challenge is to embed timing information in patterns to express complex synchronization strategies and to detect the occurrence of a pattern "in time" and not in the score.

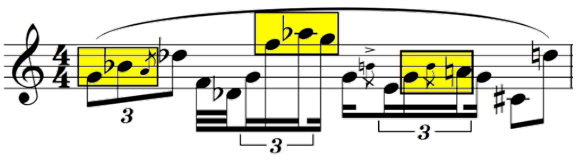


Figure 2. Occurrences (boxed) of a pattern of three consecutive notes x, y and z such that $x < y$ and $y > z > x$.

Adding the possibility to write rule $e \to a$ in addition to an *Antescofo* augmented score might seem at first sight unnecessary: because the score to follow is known *a priori*, it might seem enough to search prior any performance for the occurrence of the pattern e (as illustrated in Fig. 2) and to insert the action a in the augmented score. Nevertheless:

- Even if the occurrence of a pattern can be found before a performance, it can be convenient for the composer to factor out a series of actions to trigger repeatedly.
- Performance errors make the actual sequence of musical events different from the sequence specified in the score.
- The pattern may refer to the current value of a parameter specified only partially in the score (*e.g.*, the specification of the tempo in the score is relative and its true value is known only at performance time).
- The pattern may take into account parameters of the audio input that are not specified in a score (like dynamics or any signal descriptor that can be used to characterize the current audio input).
- Logical conditions may refer to the value of variables computed in the actions that cannot be statically inferred.
- And patterns make possible to specify an electronic response in the case of *improvisation* or in case of *open score* where the sequence of notes is only partially known (as in the case of non deterministic jumps between several score fragments).

These reasons motivate the introduction of temporal patterns in *Antescofo* in addition to the more classical transducer approach of score following.

For example, with temporal patterns it is possible to mimic *neumatic notations* used in Eastern and Western early musical notation to define a general shape but not necessarily the exact notes or rhythms to be produced. It is also possible to specify *open scores* [15], that is, score where the actual sequence of musical events is known only at performance time.

2. Related Works

Event-condition-actions (ECA) rules are a common formalism for the declarative specification of actions to be performed in response to events provided some condition hold. ECA rules are widespread in data warehouse and active databases systems [10, 25], business processes [21], network management [32], intrusion detection systems and monitoring application [27], real-time systems [6, 18] and in many other application fields. These different communities stress the concepts of "timeliness" and "flow processing" as a common goal but the corresponding systems differ in many aspects, including architecture, rule languages, and processing mechanisms.

In the domain of databases, the focus is certainly more put on the processing of streams of data coming from different sources. The relationships between the information items and the passing of time is mainly restricted to precedence relationships [10] and time is handled by time-stamping the events.

This is also the case for the synchronous languages approach in real-time systems, as exemplified by LUSTRE or Esterel, where powerful constructions make possible to specify concisely sophisticated precedence relationships but where time-stamping or the introduction of a periodic event (a clock) must be explicited to take into account the time elapsed between two events. The synchronous dataflow paradigm developed by LUSTRE makes explicit the stream of events as a value specified declaratively through equations. This is also the case in Functional Reactive Animation developed to specify time-varying media [14], as well as many other DSLs. However temporal regular expression have not been explicitly considered as a language construct in these contexts, with the exception of the related notion of mode automaton [23] in LUSTRE. Mode automata are used to specify several independent "running modes" but they are not used to span new actions, only to switch between set of predefined actions.

The handling of time, either discrete or quantitative, is also present in log auditing tools, in fault diagnosis and monitoring systems where the objective is to detect deviations from normal activity profiles. Some of them explicitly rely on temporal logic for the definition of the misuses to detect [7, 28]. Temporal logics have the advantage that they are a high-level and powerful notation for events occurring as time passes. The expression of temporal patterns as logical formulas transforms the pattern-matching as a problem of model-checking. This approach has the advantages of being well-founded and model-checking temporal logics is a well-studied topic. This approach is however less attractive than it may appear at first sight, for several reasons.

Antescofo temporal patterns may express quantitative properties on time, for instance to put a deadline on the waiting of an event (operator `Before` which can be used to specify that an event should occur after d time units regardless how many other events have occurred in between). This feature calls for bounded temporal operators, as in MTL (Metric Temporal Logic) [19] or TPTL (Timed Propositional Temporal Logic) [2]. But the possibility to express non-local timing requirements, through pattern variables, rules out MTL and more generally requires a fragment of first-order temporal logic.

Model-checking a propositional temporal formula uses the standard automata-theoretic model-checking algorithm [30]. For *first-order* temporal logic, this automaton would be infinite. The code generated in Sect. 6, derived from the semantics presented in Sect. 5, can be seen as constructing finite portions of this infinite automaton on demand and in real-time, for a relevant fragment.

The works on the *run-time verification* of logical formulas are more recent and addresses the problem in a real-time context. The objective is to check whether the run of a system under scrutiny satisfies or violates some correctness properties. In [4], a technique is proposed to translate a correctness property φ into a *monitor* used to check the current execution of a system. The property is expressed in timed linear time temporal logic (TLTL), a natural counterpart of LTL in the timed setting [12]. The run-time verification shares many similarities with model checking, but there are important differences: only one execution is checked (not all possible execution paths), the run is bounded (only finite traces are considered) and the techniques focus on on-line checking (considering incremental check and disallowing to make multiple passes over the sequence of events). Nevertheless, *Antescofo* temporal patterns only deal with the history of past events to produce their output, while formula in TLTL may express rules that require future information to be entirely evaluated. This leads [4] to the development of an ad-hoc three valued semantic (*true*, *false* and *don't know yet*) which is not relevant to decide if a pattern matches the prefix of a trace. The problem is that TLTL is not totally suited to the task of pattern matching: it is both too expressive and sometimes too cumbersome. *Antescofo* temporal pattern sequences lead to formulas of form $\psi \wedge \diamond\varphi$ where ψ are formulas whose validity can be decided without having to look at future events and φ are formulas of the same form. So, *Antescofo* patterns address a very limited set of TLTL formulas and only specify eventuality properties. But this set is dedicated to the concise expression of the temporal conditions that are relevant in our application domain. The need for simple formalisms when dealing with event-based requirements, instead of powerful but often cumbersome logics, has been pointed out in several works [1, 29]. For example, the specification of a pattern which matches an event e_0 followed by two events $e1$ and $e2$ (in any order) which are not separated by another event $e3$ leads to the logical formula

$$\begin{aligned}
&\mathit{after}(e_0) \Rightarrow ((\neg(\mathit{after}(e_1) \wedge \mathit{after}(e_2))) \\
&\qquad \exists\mathrm{U}(\mathit{after}(e_1) \vee (\neg\mathit{after}(e_3)\exists\mathrm{U}\,\mathit{after}(e_2)))) \\
&\qquad\qquad \wedge ((\neg(\mathit{after}(e_1) \wedge \mathit{after}(e_2))) \\
&\qquad\qquad \exists\mathrm{U}(\mathit{after}(e_2) \vee (\neg\mathit{after}(e_3)\exists\mathrm{U}\,\mathit{after}(e_1))))
\end{aligned}$$

which is neither concise nor very readable. This drawback should not prevent using a temporal logic as a back-end to define the semantics of temporal patterns. However, we prefer to give in Sect. 5 a formal semantics in a denotational style, defining explicitly the matching function on time-event sequence. This approach gives us both a reference point for understanding patterns, a direct executable specification and also paves the way for considering optimizations of the generated code, which is subject to stringent efficiency requirements, both in time and in space.

3. Brief Overview of *Antescofo* DSL

The *Antescofo* domain specific language relies partly on concepts introduced in synchronous programming languages in the field of embedded systems. It further addresses the management of dynamic duration related to the musical tempo extracted from an audio stream. As a reactive language, an *Antescofo* augmented score establishes a correspondence between the occurrence of events in the environment and actions that are triggered by this event. The occurrence of an action may also trigger some other actions.

The action language is procedural: atomic actions can be used to evaluate expressions, to launch conditionally or to delays others actions, to send messages to the external environment and to update variable values[1]. Variable identifiers start with a dollar character to distinguish it from the message receiver used to communicate with the external environment. The `@local` statement introduces local

[1] The language includes data types like boolean, string, float, vector, map. . . and also lambda expressions and processes. Lambdas are first-order values, as well as processes, which are abstractions over actions where lambdas are abstractions over expressions.

variables in compound actions. Compound actions can be used to group, to iterate or to span others actions.

The group is the simplest compound action: in a sequence $a_1\, a_2 \ldots$, the action a_{i+1} is launched right after the launching of a_i that is, "simultaneously but in the right order" [5]. Delays d can be expressed in relative time (*i.e.* relatively to the tempo of the musician during the performance) or in absolute time (wall clock time). They are used to postpone the triggering of the associated action. So for a group $a_1\ 1.5\, a_2\ a_3\ a_4\ 2\, a_5$, if a_1 occurs at date 0, then a_2, a_3 and a_4 occur at date 1.5 and a_5 at date 3.5. As usual in synchronous languages, an atomic action takes no time to be performed.

The `whenever` action is a compound action used to launch actions conditionally on the occurrence of an arbitrary logical condition. The occurrence of the condition is qualified as an "out-of-time" event because it does not explicitly appear in the event specified in the score. The action

```
whenever (cond) { actions } stop
```

becomes *active* when it is triggered and remains active until its end specified by the *stop* clause. When active, each time a variable appearing in the boolean expression *cond* is updated, *cond* is re-evaluated. We stress the fact that only the variables that appear syntactically in *cond* are tracked. If the condition evaluates to true, an instance of the body of the `whenever` is launched as a *parallel process*. Notice that several such processes can coexist at the same moment, depending on the duration of the actions in the body and the updates of the boolean condition *cond*.

The *stop* clause is optional and used to limit the temporal scope of the `whenever`. When missing, the `whenever` is active until the end of the *Antescofo* program. If *stop* is a clause `during[`n`#]`, the `whenever` becomes inactive after the condition has been evaluated n times (irrespectively of the result of the evaluation). If the clause takes the form `during[`d`]` the `whenever` will be active for a period of d time unit (implicitly, the time unit is relative to the tempo of the musician, but it is also possible to refer to wall clock time).

Antescofo variables can be updated from the external environment. A `whenever` on these variables allows *Antescofo* to react to arbitrary external conditions and extends the coupling of the reactive engine with the environment beyond the listening machine. The listening machine also updates the variable `$PITCH` representing the current pitch,`$DUR` representing the duration of the current note in the score and some other parameters (position in beat in the score, current tempo of the musician, etc.).

```
1  whenever ($PITCH) {
2    @local $x
3    $x := $PITCH
4    whenever ($PITCH > $x) {
5      @local $y
6      $y := $PITCH
7      whenever ($PITCH<$y & $PITCH>$x) {
8        @local $z
9        $z := $PITCH
10       print "Found␣one␣occurrence␣of␣P"
11     } during [1#]
12   } during[1#]
13 }
```

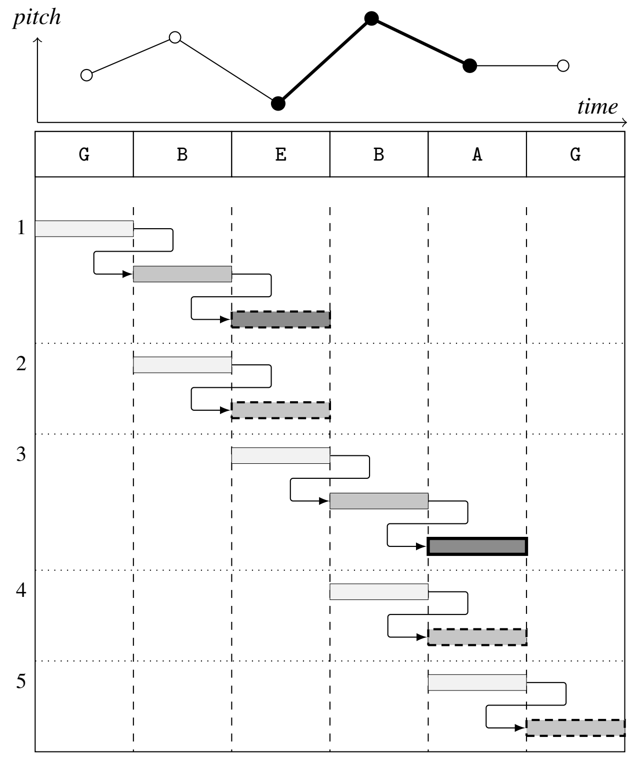


Figure 3. A fragment of *Antescofo* code that triggers action a on the reception of 3 consecutive notes x, y, z such that $x < y > z > x$. See text for explanation.

3.1 A Motivating Example

The detection of the pattern described in Fig. 2 cannot be written as a sequence of three `whenever` (they would be activated in parallel) but rather as nested `whenever`, where the triggering of an enclosing body activate a new one, see top of Fig. 3.

The behavior of this code fragment on the notification of a series of pitches (indicated in the middle of Fig. 3) is illustrated on the bottom of the same figure. This sequence of pitches contains only one occurrence of the pattern, figured in bolder line.

The `whenever` in line 1 (W1) has no stop clause. It will be active until the end of the program. The net effect is that its body is triggered each time `$PITCH` is updated (a non-zero number evaluates to true). The `whenever` at line 4 (W2) and at line 7 (W3) have a `during` clause specifying that they must be deactivated after 1 test. The activity table at bottom of Fig. 3 represents the flow of evaluation. A column is a time instant. The evaluation of the condition of (W1) is pictured in pale gray, (W2) in middle gray and (W3) in dark gray. When the evaluation returns true the border is solid, otherwise it is dashed.

On the reception of the first note, the condition of (W1) returns true. So, one instance of the body of (W1) is running now in parallel with (W1), that is, one instance of (W2) is activated and waiting for a note. The different instances of (W1) body are numbered and correspond to the row of the activity table. On the reception of the second note, this instance (row 1) evaluates to true so (W2) launches its body and one instance of (W3) is activated. The reception of the third pitch does not satisfy the condition of (W3). Meanwhile, (W1) has also been notified by the reception of the second notes which trigger one instance of (W2) body (row 2) in parallel. Etc.

Admittedly the specification of such a simple pattern is contrived to write. And it becomes even more cumbersome if one wants to manage duration and elapsing time. The objective of *Antescofo* temporal patterns is to simplify such definition. The idea is to specify a pattern elsewhere and then to use it in place of the logical condition of a `whenever`:

```
@pattern P  { ... }
...
```

```
whenever pattern::P
{ print "Found␣one␣occurrence␣of␣P" }
```

At parsing time, such whenever are recognized and translated on-the-fly into an equivalent nest of `whenever`.

4. *Antescofo* Temporal Patterns

We describe through examples the notion of temporal patterns. Their semantics is proposed in Sect. 5 and their implementation in Sect. 6.

Antescofo temporal patterns are inspired by regular expressions. An ATP P is a sequence of atomic patterns. There is no operators similar to the option operator $r?$ or the iterations operators r^* or r^+ available for a regular expression r. The reason is that ATP matching must be done in real-time and must be causal: the decision that a pattern matches must be done *with the last atomic event matched by the pattern*, as soon as it occurs. This is not the case for example with r^+ which need to look one token ahead to determine the subsequence matched.

There are two kinds of atomic patterns: `Event` that corresponds to a property satisfied on a time point and `State` corresponding to a property satisfied on a time interval.

4.1 Event Patterns

A pattern `Event $X` matches an update of the variable `$X`. This variable is said *tracked* by the pattern. Three optional clauses can be used to constraint the matching: `value`, `where` and `at`. The `value` clause constrains the value of the tracked variable. For example:

```
Event $PITCH value G4
```

matches only when `$PITCH` is assigned to `G4`. The `where` clause is used to specify a guard with an arbitrary boolean expression: the guard is evaluated at matching time and the matching fails if it evaluates to false. The boolean expression can be any valid *Antescofo* expression and may refer to arbitrary variables. The `at` clause is used to constraint the date of matching.

Pattern Variables. Pattern variables can be used to match and to record some parameters of the matching. Pattern variables are declared at the beginning of a pattern definition with a `@local` statement and can then be used elsewhere in the pattern expressions. For example, the pattern described in paragraph 3.1 becomes:

```
@local $x, $y, $z
Event $PITCH value $x
Event $PITCH value $y where $x < $y
Event $PITCH value $z
  where ($y > $z) & ($z > $x)
```

Pattern variables can be used in the pattern clauses. For example:

```
@pattern Twice {
  @local $x
  Event $PITCH value $x
  Event $PITCH value $x
}
```

matches two consecutive updates of variable `$PITCH` with the same unknown value referred by `$x`: *local variables appear as constraints* linking the patterns.

However, not all constraint are accepted: only syntactic matching as time progress is used to resolve the constraints expressed through the pattern variables. This restriction ensure that the matching is causal. For example, a pattern like

```
@local $x, $y
Event $PITCH value ($x + $y)
Event $PITCH value $x + 2*$y
```

is rejected at parsing time by *Antescofo* because the constraint between the values of the first and second event is an equation that cannot be solved by syntactic substitution as the time progress (in the example, we have to wait the second update of `$PITCH` to decide if the first pattern has matched the first update).

The constraint accepted in ATP have a simple operational interpretation. Consider pattern `Twice`: when the first event is matched, a value is given to the pattern variable `$x`. When the second event is matched, this value is used to constrain the match. This record-then-match behavior is just the operational explanation of the existential quantification in logic formula when no unification nor solver are available, only matching following the patterns order, as in ML-like pattern-matching [22].

The scope of the pattern variables extend to the actions triggered by the pattern, when they can be used as ordinary variables. For example:

```
@pattern P {
  @local $t
  Event $PITCH at $t
}
...
whenever pattern::P
{ print "found␣a␣P␣at␣" $t }
```

will report the date of the matching for each occurrence of the pattern.

Tracking Multiple Variables Simultaneously. It is possible to track several variables simultaneously: the pattern matches when one of the tracked variables is updated (and if the other clauses are fulfilled). For instance, to match an update of `$X` followed by an update of either `$X` *or* `$Y` before 1.5 beat, we can write:

```
@local $t1, $t2
Event $X at $t1
Event $X, $Y at $t2
  where ($t2 - $t1) < 1.5
```

4.2 Temporal Scope and the `Before` Clause

The previous example shows that timed properties can be expressed relying on the `at` and the `where` clause. It is however not easy to express that a variable must take a given value within the next three updates. This drawback motivates the introduction of the `Before` clause to specify the *temporal scope* on which a matching is searched.

When *Antescofo* is looking to match the pattern `Event $X`, the variable `$X` is tracked right after the match of the previous pattern. Then, at the *first value change* of `$X`, *Antescofo* check the various constraints of the pattern. If the constraints are not met, the matching fails. The `Before` clause can be used to shrink or to extend the temporal interval on which the pattern is matched beyond the first value change. For instance, the pattern

```
@pattern TwiceIn3B {
  @local $v
  Event $V value $v
  Before[3] Event $V value $v
}
```

is looking for two updates of variable `$V` for the same value `$v` within 3 beats. *Nota bene* that other updates for other values may occurs as well as updates for `$V` but, for the pattern to match, variable `$V` must be updated for the same value before 3 beats have elapsed from the match of the first event.

If the temporal scope `[3]` is replaced by a logical count `[3#]`, we are looking for an update for the same value that occurs in the next 3 updates of the tracked variable. The temporal scope can also be specified in seconds.

The temporal scope defined on an event starts with the preceding event. So a `Before` clause on the first `Event` of a pattern sequence is meaningless and actually forbidden by the syntax.

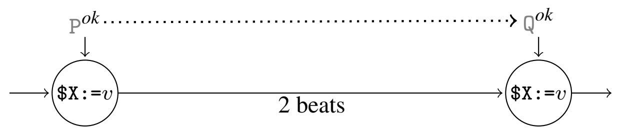
4.3 State Patterns

The `Event` pattern corresponds to a logic of *instants*: each variable update is meaningful and a property is checked on *a given point in time*. This contrasts with a logic of *states* where a property is looked *on an interval of time*. The `State` pattern can be used to face such case.

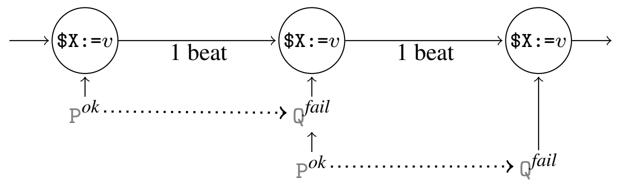
A Motivating Example. Suppose we want to trigger an action when a variable `$X` takes a given value v for at least 2 beats. The pattern sequence

```
@Local $start, $stop
Event $X value v at $start          P
Event $X value v at $stop         }
  where ($stop - $start) >= 2     } Q
```

does not work: it matches two successive updates of `$X` that span over 2 beats:



but it would not match three consecutive updates of `$X` for the same value v, one at each beat, a configuration that should be recognized:



It is not an easy task to translate the specification of a state that lasts over an interval into a sequence of instantaneous events, because they can be an arbitrary number of events that does not change the state, while the `Event` pattern matches exactly one event.

The `State` pattern make the previous constraint easy to specify:

```
State $X where ($X == v) during[2]
```

matches an interval of 2 beats where the variable `$X` constantly has the value v (irrespectively of the number of variable updates).

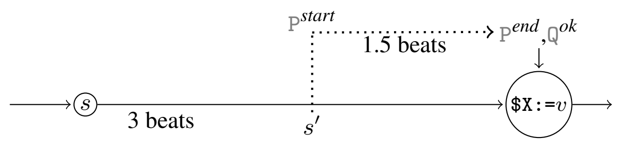
Four optional clauses can be used to constraint a `state` pattern: `Before` and `where` clauses constrain the matching as described for `Event` patterns. The `at` clause is replaced by the two clauses `start` and `stop` to record or constrain the date at which the matching of the pattern has started and the date at which the matching stops. There is no `value` clause because the value of the tracked variable may change during the matching of the pattern, for instance when the state is defined as "being above some threshold". The `where` clause may refer to a pattern variable set in the `start` clause, but not to the value of a `stop` clause because the date at which the pattern ends is known only in the future. The `during` clause can be used to specify the duration of the state, *i.e.* the time interval on which the various constraints of the pattern must hold. If the specified constraints are not satisfied, the matching fails but, if there is a `Before` clause, a new attempt is launched at each update of the tracked variable, until the expiration of the before clause.

"Discrete"* vs. *"Continuous" State Properties. Contrarily to `Event`, the `State` pattern is not driven solely by the updates of the tracked variables: in addition, the constraints are also checked when the matching of a `State` is initiated. Furthermore, the matching of a `State` stops when the specified duration has elapsed, independently of the variables update. If there is no `during` clause, the pattern tracks the variables whilst the constraints are satisfied and the matching stops as soon they are no longer satisfied.

Still, it remains that checking the guard of a state is done on discrete time instants (corresponding to the occurrence of a variable update eventually delayed by durations taken in a finite set). This constrains the kind of properties that can be handled to the properties that can be expressed relatively to a given countable set of dates (in continuous time) and set aside arbitrary properties defined on continuous time. Consider for example the pattern

```
State $X where true during[1.5]
Event $X where ($X == v)
```

with `$X` updated at date 0 and assigned to v at date 3. We suppose further that the pattern matching starts at date $s = 0$. With these assumptions, one may consider that there is a match M starting at date $s' = 1.5$ and ending at date 3:



Nevertheless *Antescofo* does not report any match because there is no event at date 1.5 that can be a possible start to match the `State` pattern. The arbitrary date $s' = 1.5$ does not belong on the set of dates on which the pattern properties are checked.

One may wonder if the notion of state can be extended to handle such examples. For instance, in the FRAN framework [14] it is possible to express arbitrary equations on the date of an event in the specification of this event. Interval analysis is then used to solve numerically the equations. This approach can be used to extend the kind of constraints expressible in a `where` clause but does help here: the constraint on the start date of the `State` pattern implies the date of a future event not yet known. As a matter of fact, reporting the match M would imply either: a) to start the matching at each time instant of the continuous time, which is not reasonable, or, b) to access all the past states of the system to check the `State` pattern *a posteriori* when the `Event` pattern occurs. The approach (b) implies an unbounded memory and cannot be extended to patterns that do not end with an event.

Example. We illustrate the `State` construction with a pattern used to characterize some kind of "non monotonic increase" of a signal:

```
State $X during[a] where $X > b
Before[c] State $X where $X > d
```

The corresponding behavior is sketched in Fig. 4.

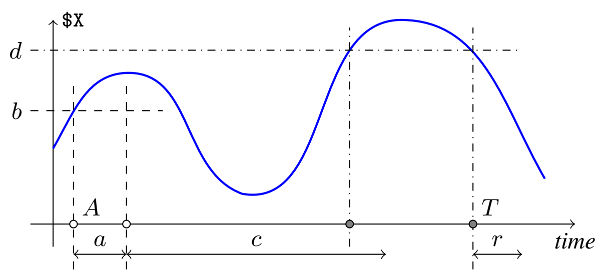


Figure 4. Matching two successive states, the first above level b with a specified duration of a and the second above level d with no duration and within a temporal scope of c. See text for explanations.

The diagram assumes that variable `$X` is sampling at a rate δ the underlying continuous evolution of a signal f. The first `State` pattern is looking for an interval of length a where constantly variable `$X` is greater than b. The first possible interval start at date A and is figured by the two white circles on the time axis. The second `State` pattern must start to match before c beats have elapsed since the end of the previous pattern. The match starts as soon as `$X` is greater than d. There is no specification of a duration for the second state, so it finishes its matching at time T as soon as `$X` becomes smaller than d. The matched interval is marked with the two dark circles on the time line.

4.4 Limiting the Number of Matches

The same pattern may match distinct occurrences that start or that stop at the same time instant. This behavior may be unwanted because it will produce "spurious matches" that reach, by multiple paths, the same time point T.

The "Earliest Match" Property. A regular expression may match several prefixes of the same string. For example, $a.b^*$ matches the three prefixes a, ab, abb of the word abb. Usually, a pattern matcher reports only one match, characterized by an additional property, *e.g.*, "the longest match".

A similar problem exists for temporal patterns, even in the absence of iteration operators: several distinct occurrences of the same pattern starting at the same date but ending a different date may exists. Such alternative solutions may appear when the temporal scope of a pattern is extended beyond the first value change: then, distinct matches within the temporal scope may satisfy the various constraints of the pattern[2]. For instance, consider the pattern `TwiceIn3B` in paragraph 4.2. If the variable `$V` takes the same value three times within 3 seconds, say at the dates $t_1 < t_2 < t_3$, then `TwiceIn3B` occurs three times as (t_1, t_2), (t_1, t_3), and (t_2, t_3).

To ensure the real-time decidability of the matching, the occurrence (t_1, t_2) of the match must be reported because at t_2 there is no information about a possible further match. So the question is to decide if further matches have to be reported or not. We adopted the common behavior of reporting only one match (this is for instance the behavior of `lex` or `grep`).

In other word, the *Antescofo* pattern matching stops looking for further occurrences in the same temporal scope, after having found the first one. This behavior is called the *earliest match* property. In the previous example, with this property, only the two matches (t_1, t_2) and (t_2, t_3) are reported.

The Refractory Period. Symmetrically, several occurrences of the same pattern may start at distinct time points to end on the same time point. For instance, the curve sketched in Fig. 4 presents many other possible occurrences of the pattern that finishes at instant T. These occurrences start at $A + n\delta$, where δ is the sampling rate of the curve (*i.e.*, the rate at which `$X` is updated), as long as $f(A + n\delta + x) > b$ for $x \in [0, a]$.

In such case, a `@refractory` period can be used to restrict the number of successful (reported) matches. The `@Refractory` clause specifies the period after a successful match during which no other match may occur. This period is counted starting from the end of the successful match. A possible refractory period r is represented in Fig. 4. The refractory period is defined for a pattern sequence, not for an atomic pattern. The `@Refractory` clause must be specified at the beginning of the pattern sequence just before or after an eventual `@Local` clause. If there is no refractory period specified, all feasible paths trigger the action.

[2] If there is no `Before` clause, the temporal scope is "the first value change" which implies that there is *at most* one match.

4.5 Patterns Hierarchization

Because atomic patterns track ordinary *Antescofo* variables, it is very easy to create patterns P for more complex events and states from more elementary patterns Q. The idea is to update with Q a variable which is then tracked by P. For instance, suppose that patterns $G_1, \ldots, G_4$ match some basic gestures reported through the updates of some variables. Then, the recognition of a sequence G_{seq} of gestures $G_1 \cdot (G_2|G_3) \cdot G_4$, *i.e.* G_1 followed either by G_2 or G_3 followed by G_4, is easily specified as:

```
$g := 0
whenever pattern::G1 { $g := 1 }
whenever pattern::G2 { $g := 2 }
whenever pattern::G3 { $g := 3 }
whenever pattern::G4 { $g := 4 }

@pattern Gseq {
  Event $g value 1
  Event $g where ($g==2) || ($g==3)
  Event $g value 4
}
...
whenever pattern::Gseq { ... }
```

5. *Antescofo* Temporal Patterns Semantics

We present in this section a simplified pattern-matching algorithm for *Antescofo* temporal patterns, following a denotational style. We first introduce the notion of time-event sequence which formalizes the input stream on which the matching is done. Then we define the matching of a pattern P on a time-event sequence S by a function which returns either the time at which the matching succeeded (from the start of S) or *fail*. This function is defined by induction on both P and S.

5.1 Time-Event Sequences

It should be clear by now that the *Antescofo* DSL goes beyond the synchronous stream of atomic events, to handle the metric passage of time. This leads to the notion of *time-event sequence* representing an interleaving of time passages and events [3]. As usual in synchronous languages, an event is atomic: updates of a variable occur at certain time points and consume no time. Time-event sequences allow two events to happen simultaneously but still one after the other. It is very convenient to have events and actions that can happen at the same metric time instant, but in some well definite order. For example, on some event, an audio filter must be turned on and then it must receive some control parameters. Obviously, the control parameters must be sent only when the filter is on, but it is pointless to explicitly wait some arbitrary small delay between the two actions.

We formalize time-event sequences as follows. We represent the time passage by an element of $\mathbb{R}^+$. The elements of $\mathcal{U}$, the set of events, are the updates of the variables: an element of $\mathcal{U}$ is a term $x := v$ where $x \in \mathcal{I}$ is an *Antescofo* variable and $v \in \mathcal{V}$ an *Antescofo* value. We look at these sets as flat domains $\mathcal{U}_\perp$ and $\mathbb{R}^+_\perp$ with the same minimal element $\perp$: all elements except $\perp$ are incomparable [24] for the ordering $\preceq$ (this order is the domain order and should not be confused with the numerical order $\leq$ on $\mathbb{R}^+$). So, a time-event sequence is an element of the monoid

$$\mathcal{S} = (\mathcal{U}_\perp \cup \mathbb{R}^+_\perp)^* \,/\, \sim$$

where the monoid operation is denoted by $\cdot$ and where $\sim$ is the congruence relation defined by:

$$d \cdot d' \sim d + d', \quad 0 \cdot s \sim s, \quad s \cdot 0 \sim s$$

for $d, d' \in \mathbb{R}^+$ and $s \in \mathcal{S}$. The congruence relation is used to aggregate consecutive time passages and to throw away useless time passages of duration zero: time passages are indecomposable and bounded by events. This monoid equipped with the prefix order $\preceq$, *i.e.* $s \preceq s'$ iff it exists t such that $s' = s \cdot t$, is a domain. The empty element of $\mathcal{S}$ is denoted by ϵ.

5.2 The Patterns

Without loss of generality, we restrict ourselves to the case of patterns tracking only one variable. We assume also that the argument of the clauses `at`, `value`, `start` and `stop` is always a fresh pattern variable (that is, a pattern variable that do not appear in a previous clause). Furthermore, for an `Event` pattern, we assume that all the clauses appear. For a `State` pattern, we assume that there is a `Before` clause and no `start` and `stop` clauses. The management of these clauses is similar to the `value` clause of an `Event` as done in the equations of Fig. 5 and presents no difficulty. If a pattern is the first of the sequence, the value of its `Before` clause is $+\infty$ by convention.

As a matter of fact, a pattern can always be rewritten into an equivalent pattern that fulfills these assumptions. For example

```
Event $X value v at $s where e
Event $Y value v' at $s where e'
```

where v, v' are expressions, can be rewritten in

```
Event $X value $v at $s
  where e && ($v == v)
Event $Y value $w at $t
  $where e' && ($w == v') && ($s == $t)
```

where `$v`, `$w`, `$t` are fresh identifiers.

An implicit `Before` clause corresponds to a temporal scope of "first value change", so

```
Event $X ...
Event $Y ...
```

can be rewritten in

```
Before[+∞] Event $X ...
Before[1#] Event $Y ...
```

And a pattern tracking two variables `$X` and `$Y`

```
Event $X, $Y ...
```

can be rewritten into an equivalent program using a fresh variable `$XY` to track the updates of both `$X` and `$Y`.

```
whenever ($X==$X || $Y==$Y) { $XY:=true }
...
Event $XY ...
```

Here, the expression `$X==$X` is used to have a predicate which is always true on the update of `$X`.

With these assumptions, a pattern is a sequence of `Event` and `State`:

$$
\begin{aligned}
\mathcal{P} &::= \varepsilon \mid \textit{Event} \cdot \mathcal{P} \mid \textit{State} \cdot \mathcal{P} \\
\textit{Event} &::= \texttt{Before[}\ \textit{Dur}\ \texttt{]}\ \texttt{Event}\ \mathcal{I}\ \texttt{at}\ \mathcal{I}\ \texttt{value}\ \mathcal{I}\ \texttt{where}\ \textit{Exp} \\
\textit{State} &::= \texttt{Before[}\ \textit{Dur}\ \texttt{]}\ \texttt{State}\ \mathcal{I}\ \texttt{where}\ \textit{Exp} \\
&\quad\mid \texttt{Before[}\ \textit{Dur}\ \texttt{]}\ \texttt{State}\ \mathcal{I}\ \texttt{where}\ \textit{Exp}\ \texttt{during[}\ \overline{\mathbb{R}^+}\ \texttt{]} \\
\textit{Dur} &::= \overline{\mathbb{R}^+} \mid \mathbb{N}\texttt{\#}
\end{aligned}
$$

where ε is the empty pattern sequence, *Exp* is the (unspecified) set of *Antescofo* expressions, $\overline{\mathbb{R}^+} = \mathbb{R}^+ \cup \{+\infty\}$ and $d < +\infty$ for all $d \in \mathbb{R}^+$. The notation P_x is used to make explicit the variable x tracked by the pattern P.

5.3 The Pattern Matching Function

An *environment* $\rho \in \mathcal{E}$ is a partial function from the set of variables $\mathcal{I}$ to the set of values $\mathcal{V}$. The *augmentation* $\rho[\texttt{\$X} := v]$ of an environment ρ with identifier `$X` and value v is a new environment ρ' such that $\rho'(\texttt{\$X}) = v$ and $\rho'(x) = \rho(x)$ for all $x \neq \texttt{\$X}$. We write $\rho[x_1 := v_1, x_2 := v_2, \dots]$ as a shorthand for $\big(\rho[x_1 := v_1]\big)[x_2 := v_2, \dots]$ and $\rho[x \mathrel{+}= d]$ as an abbreviation for $\rho\big[x := \rho(x) + d\big]$. We also reserve the identifier `$NOW` to record the "current time" in the environment for some bookkeeping.

Let $\mathbf{E} : \textit{Exp} \to \mathcal{E} \to \mathcal{V}$ be the function used to evaluates an expression $e \in \textit{Exp}$: $\mathbf{E}\llbracket e \rrbracket \rho$ returns the value of the expression e in the environment ρ. The two booleans *true* and *false* belong to $\mathcal{V}$. We do not specify the function $\mathbf{E}$ in this article but its definition is standard.

Let P be a pattern sequence. We define the matching of P on a time event sequence S by a function $\mathbf{M}$:

$$\mathbf{M} : \mathcal{P} \to \mathcal{E} \to \mathcal{S} \to \mathbb{R}^+ \cup \{\textit{fail}\}$$

specified inductively by the equations on Fig. 5. If P matches a prefix of S, the function $\mathbf{M}$ returns a date, else it returns *fail*. The date returned in case of success is the date at which the action triggered by the pattern must be launched (*i.e.*, the `at` date for an `Event` and the `stop` date for a `State` pattern of the last atomic pattern of the sequence). This date is the earliest possible match, thus satisfying the *earliest match* property. We do not model here the mechanism of refractory period, which is straightforward by recording the history of matches, nor the semantics of actions, which is out of the scope of this paper[3].

The basic idea is to define by case analysis what happens on the reception of an event or when the time is passing. In this sense, the $\mathbf{M}$ function is similar to the Brzozowski's derivatives of a regular expression [8]. Our context at the same time is simpler (there is no iteration operator and there is no need to represent symbolically the derivatives in a closed form) and presents specific difficulties (the handling of both event and time passage, and the management of variables). We follow the approach already taken in [17] by augmenting the derivatives with an environment.

The equations of Fig. 5 are commented below. These equations are well formed recursive definitions: the left hand sides specify mutually disjoint cases, and the right hand sides are composition of continuous functions on domains. So they admit a least fixed point which is the denotation of a pattern: a function which, given an environment and a time-event sequence, returns the date of the earliest match or *fail*.

General Equations. The matching always succeeds if the pattern is the empty sequence (Eq. 1). Symmetrically, the matching fails if the input timed-event sequence is exhausted but there is still an atomic pattern to process (Eq. 2). And (Eq. 3) expresses the matching is insensitive to the update of a variable that is not the tracked variable. The remaining equations correspond to a definition by case on the structure of the first pattern of the sequence (the rest of the sequence is denoted by Q).

Matching an **`Event`.** The matching of an `Event` pattern is defined by (Eq. 4–7). The notation P_x is used to factorize the writing of the `Event` through the various possible value for the `Before`

[3] It would require transformations of the time-event sequence in the right hand side of the equations in Fig. 5 beyond taking its tail. To take into account causality, this imply to rely on $(\mathcal{U}_\perp \cup \mathbb{R}^+_\perp)^\$ / \sim$ for the time-event sequences where $\mathcal{X}^\$ \cong \mathcal{X} \otimes \mathcal{X}^\$_\perp$ which differs from the domain of streams in that the former does not allow $\perp$ components to be followed by non-$\perp$ components. This domain makes the handling of temporal shortcut, *i.e.* a pattern that launches an action which leads to trigger the same pattern in the same time instant, more simple.

$$
\begin{array}{ll}
(1) & \mathbf{M}[\![\varepsilon]\!]\rho\, S = \rho(\texttt{\$NOW}) \\
(2) & \mathbf{M}[\![P]\!]\rho\,\epsilon = \mathit{fail}, \qquad P \neq \varepsilon \\
(3) & \mathbf{M}[\![P_x \cdot Q]\!]\ \rho\ \big(x' :\!= v \cdot S\big) = \mathbf{M}[\![P_x \cdot Q]\!]\ \rho[x' :\!= v]\ S \quad \text{where } x \neq x'
\end{array}
$$

let $P_x =$ `Event` x `at` y `value` z `where` e **in:**

$$
\begin{array}{ll}
(4) & \mathbf{M}[\![\texttt{Before[}d\texttt{]}\ P_x \cdot Q]\!]\ \rho\ (d' \cdot S) = \begin{cases} \mathit{fail}, & \text{if } d \leq d' \\ \mathbf{M}[\![\texttt{Before[}d-d'\texttt{]}\ P_x \cdot Q]\!]\ \rho[\texttt{\$NOW} \mathrel{+}= d']\ S, & \text{if } d > d' \end{cases} \\
(5) & \mathbf{M}[\![\texttt{Before[}0\texttt{\#]}\ P_x \cdot Q]\!]\ \rho\ S = \mathit{fail} \\
(6) & \mathbf{M}[\![\texttt{Before[}n\texttt{\#]}\ P_x \cdot Q]\!]\ \rho\ (d' \cdot S) = \mathbf{M}[\![\texttt{Before[}n\texttt{\#]}\ P_x \cdot Q]\!]\ \rho[\texttt{\$NOW} \mathrel{+}= d']\ S \\
(7) & \mathbf{M}[\![\texttt{Before[}D\texttt{]}\ P_x \cdot Q]\!]\ \rho\ \big(x :\!= v \cdot S\big) = \begin{cases} \mathbf{M}[\![P'_x \cdot Q]\!]\ \rho'\ S, & \text{if } \mathbf{E}[\![e]\!]\rho'' = \mathit{false} \\ \min\big(\mathbf{M}[\![P'_x \cdot Q]\!]\ \rho'\ S,\ \ \mathbf{M}[\![Q]\!]\ \rho''\ S\big) & \text{if } \mathbf{E}[\![e]\!]\rho'' = \mathit{true} \end{cases}
\end{array}
$$

$$
\text{where } \rho' = \rho[x :\!= v] \text{ and } \rho'' = \rho'[y :\!= \rho(\texttt{\$NOW}), z :\!= v] \text{ and } P'_x = \begin{cases} \texttt{Before[}d\texttt{]}\ P_x, & \text{if } D = d \\ \texttt{Before[}(n-1)\texttt{\#]}\ P_x, & \text{if } D = n\texttt{\#} \end{cases}
$$

let $P_x =$ `State` x `where` e **and** $\overline{P}_x \in \{P_x,\ P_x\,\texttt{during[}d\texttt{]}\}$ **in:**

$$
(8) \quad \mathbf{M}[\![\texttt{Before[}d\texttt{]}\ \overline{P}_x \cdot Q]\!]\ \rho\ (d' \cdot S) = \begin{cases} \mathit{fail} & \text{if } d \leq d' \ \wedge\ \mathbf{E}[\![e]\!]\rho = \mathit{false} \\ \mathbf{M}[\![\texttt{Before[}d-d'\texttt{]}\ \overline{P}_x \cdot Q]\!]\ \rho'\ \cdot S & \text{if } d > d' \ \wedge\ \mathbf{E}[\![e]\!]\rho = \mathit{false} \\ \mathbf{M_S}[\![\overline{P}_x \cdot Q]\!]\ \rho\ \big(d' \cdot S\big) & \text{if } d \leq d' \ \wedge\ \mathbf{E}[\![e]\!]\rho = \mathit{true} \\ \min\begin{pmatrix} \mathbf{M_S}[\![\overline{P}_x \cdot Q]\!]\ \rho\ (d' \cdot S), \\ \mathbf{M}[\![\texttt{Before[}d-d'\texttt{]}\ \overline{P}_x \cdot Q]\!]\ \rho'\ \cdot S \end{pmatrix} & \text{if } d > d' \ \wedge\ \mathbf{E}[\![e]\!]\rho = \mathit{true} \end{cases}
$$

$$
\text{where } \rho' = \rho[\texttt{\$NOW} \mathrel{+}= d']
$$

$$
(9) \quad \mathbf{M}[\![\texttt{Before[}d\texttt{]}\ \overline{P}_x \cdot Q]\!]\ \rho\ (x :\!= v \cdot S) = \begin{cases} \mathbf{M}[\![\texttt{Before[}d\texttt{]}\ \overline{P}_x \cdot Q]\!]\ \rho'\ S & \text{if } \mathbf{E}[\![e]\!]\rho' = \mathit{false} \\ \min\begin{pmatrix} \mathbf{M_S}[\![\overline{P}_x \cdot Q]\!]\ \rho'\ \cdot S, \\ \mathbf{M}[\![\texttt{Before[}d\texttt{]}\ \overline{P}_x \cdot Q]\!]\ \rho'\ \cdot S \end{pmatrix} & \text{if } \mathbf{E}[\![e]\!]\rho' = \mathit{true} \end{cases}
$$

$$
\text{where } \rho' = \rho[x :\!= v]
$$

$$
\begin{array}{ll}
(10) & \mathbf{M_S}[\![\overline{P}_x \cdot Q]\!]\ \rho\ \epsilon = \mathit{fail} \\
 & \mathbf{M_S}[\![\overline{P}_x \cdot Q]\!]\ \rho\ (x' :\!= v \cdot S) = \mathbf{M_S}[\![\overline{P}_x \cdot Q]\!]\ \rho[x' :\!= v]\ S \quad \text{where } x \neq x' \\
(11) & \mathbf{M_S}[\![P_x \cdot Q]\!]\ \rho\ (d' \cdot S) = \mathbf{M_S}[\![P_x \cdot Q]\!]\ \rho[\texttt{\$NOW} \mathrel{+}= d']\ S \\
 & \mathbf{M_S}[\![P_x \cdot Q]\!]\ \rho\ (x :\!= v \cdot S) = \begin{cases} \mathbf{M_S}[\![P_x \cdot Q]\!]\ \rho[x :\!= v]\ S & \text{if } \mathbf{E}[\![e]\!]\rho[x :\!= v] = \mathit{true} \\ \mathbf{M}[\![Q]\!]\ \rho[x :\!= v]\ S & \text{if } \mathbf{E}[\![e]\!]\rho[x :\!= v] = \mathit{false} \end{cases} \\
(12) & \mathbf{M_S}[\![P_x\,\texttt{during[}d\texttt{]}\ \cdot Q]\!]\ \rho\ (d' \cdot S) = \begin{cases} \mathbf{M}[\![Q]\!]\ \rho[\texttt{\$NOW} \mathrel{+}= d]\ (d' - d \cdot S) & \text{if } d \leq d' \\ \mathbf{M_S}[\![P_x\,\texttt{during[}d-d'\texttt{]}\ \cdot Q]\!]\ \rho[\texttt{\$NOW} \mathrel{+}= d']\ S & \text{if } d > d' \end{cases} \\
 & \mathbf{M_S}[\![P_x\,\texttt{during[}d\texttt{]}\ \cdot Q]\!]\ \rho\ (x :\!= v \cdot S) = \begin{cases} \mathit{fail} & \text{if } \mathbf{E}[\![e]\!]\rho[x :\!= v] = \mathit{false} \\ \mathbf{M_S}[\![P_x\,\texttt{during[}d\texttt{]}\ \cdot Q]\!]\ \rho[x :\!= v]\ S & \text{if } \mathbf{E}[\![e]\!]\rho[x :\!= v] = \mathit{true} \end{cases}
\end{array}
$$

Figure 5. Specification of the *Antescofo* temporal pattern matching function $\mathbf{M}$. In these equations: $d \in \overline{\mathbb{R}^+}$; $d' \in \mathbb{R}^+$; x, x', y and z are elements of $\mathcal{I}$; P, Q are elements of $\mathcal{P}$ and P_x, P'_x are patterns, or parts of a pattern, tracking the variable x; $\rho \in \mathcal{E}$; $v \in \mathcal{V}$; $n \in \mathbb{N}$ and $n \neq 0$; $D \in \mathit{Dur}$, *i.e.*, $D = d$ or $D = n\texttt{\#}$; and $S \in \mathcal{S}$. The function $\min$ is the usual function on $\mathbb{R}^+$ extended such that $\min(d, \mathit{fail}) = \min(\mathit{fail}, d) = d$ and $\min(\mathit{fail}, \mathit{fail}) = \mathit{fail}$. The auxiliary function $\mathbf{M_S}$ has the same signature as $\mathbf{M}$.

clause. (Eq. 4) specifies the effect of the time passage on a temporal scope defined by a metric interval: if the passage of time exceeds the temporal scope, the matching fails. If the passage of time is smaller than the temporal scope, then both the temporal scope and the notion of current time are updated accordingly. If the temporal scope is a logical count (of variable updates), then the passage of time has no effect on the matching (Eq. 6). But if the logical count is exhausted, the matching fails (Eq. 5). When the tracked variable is updated, (Eq. 7), the guard of the pattern is evaluated in an environment where the tracked variable has its new value. If the result is false, this event cannot match the pattern and the matching is resumed on the rest of the sequence, with the environment updated by the new value of the tracked variable. If the guard evaluates to true, they are two possibilities: accepting this event as the event matching the pattern, or delaying the acceptation to a future event. These two possibilities are the argument of the function $\min$ in the right hand side of (Eq. 7). The function $\min$ returns the earliest date of the potential matches ($fail$ is defined as a neutral element of $\min$ to accommodate the possible mismatches).

Starting the Matching of a `State`. The matching of a `State` pattern is defined by (Eq. 8–9) with the help of the auxiliary function $\mathbf{M_S}$ defined in (Eq. 10–12). The notation P_x is used to factorize the writing of the `State` through the various possible values for the `Before` clause. $\overline{P}_x$ represents a P_x statement optionally completed by a `during` clause.

(Eq. 8) specifies the passage of time on a `State` that has not yet been triggered. The guard is evaluated at the beginning of this time passage. The property of a state must be true when "entering" in the state and remains true through the events, until the "exit" of the state. So, if the guard is false, and if the elapsed time exceeds the temporal scope, the matching fails (first case of (Eq. 8)). If the guard is false but the elapsed time does not exhaust the temporal scope, the matching resumes with the same pattern, but with an updated temporal scope to reflect the time passage (second case of (Eq. 8)). If the guard of the pattern evaluates to true, then two cases are to be considered. If the passage of time exceeds the temporal scope, then the only alternative is to accept the current time instant as the start of the matching, which is then handled by the function $\mathbf{M_S}$ (because the next input event falls outside the temporal scope and so cannot be an admissible starting point). If the temporal scope is greater than the passage of time, then we can either posit the hypothesis of the beginning of the matching, or delay it to the next event. Hence the two arguments of the `min` function in the last case of (Eq. 8). The update of the tracked variable is specified by (Eq. 9): two cases are considered following the evaluation of the guard. If the guard is not fulfilled, the matching resumes with the environment updated. If the guard is satisfied, as before we can accept this matching or delay it to a future event (the two cases of the $\min$ operator).

Finishing the Matching of a `State`. The function $\mathbf{M_S}$ is used to manage the duration specified by the optional `during` clause. It is defined by case through (Eq. 10–12). Once a `State` has started, it fails if there is no more events, nor time to finish it, (Eq. 10). And the update of a non-tracked variable has no effect, except the update of the environment. A `State` without `during` clause finishes as soon as its guard becomes false. The guard is evaluated on event only. This behavior is specified by (Eq. 11). When to stop the matching of a `State` with duration is defined by (Eq. 12). The first equation gives the effect of the passage of time and the second one, the effect of an event. On an event, the guard is re-evaluated and, if false, the matching fails.

The semantic equations of Fig. 5 make together an executable specification of the matching. This specification is however not very efficient. For instance, in (Eq. 7,8,9) the $\min$ function selects the earliest matching after the completion of both branches. An efficient on-line implementation will cut the concurrent threads of matching as soon as one solution has been found.

6. Compilation

We have developed a full prototype of ATP matching by translating the temporal patterns in a series of nested `whenever`. As a matter of fact, the body of a `whenever` is launched in parallel with the other computations on the reception of an event achieving a kind of process call, hence, a kind of function call. It is also possible to launch a group of actions after some delays and to stop a `whenever` after some duration, which corresponds to primitives making possible the online handling of the time passage. So, the idea is to translate the equations of Fig. 5 defining the function $\mathbf{M}$ into an online version using real-time processes through `whenever` and `delays`. The explicit environment used by the definition of $\mathbf{M}$ is implemented using local variables.

The function $\mathbf{M}$ is defined inductively on the sequence of atomic patterns. A rapid inspection shows that the nested calls grow exponentially with the size of the pattern sequence because of (Eq. 7,8,9). But this exponential growth is only apparent. Indeed, in term of processes, one can notice that the father processes (the left hand side of the equations) are waiting for the result returned by the recursive call on the right hand sides. So it is possible to rephrase them in order to avoid the spawn of a son. The net effect is that the generated code is a static nest of `whenever` where the number of nestings is the number of atomic patterns in the sequence. Each `whenever` can be seen as the state of an automaton waiting an event or the passage of time to go into another state. But this automaton is unfolded in real-time by the body of the `whenever` instantiated at each occurrence of an event or the time passage.

Code Template for an `Event`. The code generated for an `Event` is straightforward. Suppose that the pattern

`Before[` D `] Event` x `at` y `value` v `where` $e \cdot P$

is the first of the pattern sequence. Then, it translates into the template:

```
$last_matching := −∞
whenever (x==x) {
  @Local $continue, y, v
  $continue := true
  v := x
  y := $NOW
  if (e) { P }
}
```

Note that the expression e is evaluated after the initialization of the pattern variable y and v because e may refer to these variables. The variable `$last_matching` is used to manage the refractory time and will record the date of the last successful match in the code generated for the last pattern of the sequence. The local variable `$continue` is used to abort the `whenever` spanned by the hierarchy when the first match found. The condition x==x is a boolean expression which triggers the `whenever` when there is an update of x. The `$continue` will disallow this triggering when false. There is no `during` clause because when the pattern sequence is activated, the matching must start on each incoming event.

For an `Event` pattern in the middle of the pattern sequence, the generated code is simpler:

```
whenever ($continue && (x==x)) {
  @Local y, v
  v := x
  y := $NOW
  if (e) { P }
} during [D]
```

The body is launched only if the variable `$continue` is true. The `during` clause stops the tracking of the variable x when the interval of time specified by D is exhausted. This does not kill the instances of the body already spanned, it only avoids the spanning of new instances.

For an `Event` pattern at the end of a pattern sequence, the code includes the management of the `$continue` variable and of the refractory period:

```
whenever ($continue && (x==x)) {
  @Local y, v
  v := x
  y := $NOW
  if (e && ($NOW - $last_match < r)) {
    $last_match := $NOW
    $continue := false
    a
  }
} during [D]
```

The constant r refers to the value of the refractory period and a to the action to launch on the recognition of the pattern sequence.

The Code for a* `State`*. Similarly to the `Event` pattern, there is a slight difference between the code generated for a `State` in the first position, in the middle or at the end of the pattern sequence. We give here only a sketch of the code generated for a `State` in the middle without considering the management of `$continue` and `$last_matching`. We do not detail the management of the clauses `start` and `stop` because it is very similar to the management of the `at` and `value` clauses for `Event`, but cumbersome.

The code for a `State` with or without duration constraint is different but in the two cases, we keep track of the `State`'s property: if it is satisfied we record the date the property became true. So,

```
Before[D] State x where e during[d]
```

is translated into

```
1  @Local $started, $halt, $start
2  $started := -1
3  $halt := ($started > 0) && ! [D]
4  ...
5  whenever (x==x) @Immediate {
6    @Local $start
7    $start := ...
8    if (e) {
9      if ($started < 0)
10     { $started := $NOW }
11     d if ($started>=0 && $start>=$started)
12           { ... }
13   } else
14     $started := -1
15 } until ($halt)
```

The `@Immediate` attribute of a `whenever` forces an additional evaluation of its guard at firing time, independently of the update of the variables present in the condition. This `whenever` maintains a variable `$started` which is positive if the property e is true and records the last date at which e goes from false to true. When e is true, a conditional is also launched with a delay d (line 11). This delay is the expected duration of the `State`. When it expires, the conditional is triggered and its body is launched only if the start time of the pattern, recorded in the variable `$start`, is posterior to the last time the property became true.

The variable `$halt` play a role similar to `$continue` and is used to abort the `whenever` when the time goes outside the specified temporal scope. The computation of the expression $[D]$ is not figured here (it implies several auxiliary variables to record the number of updates of the tracked variable or of the elapsed time). Such optimization is needed to dispose efficiently the instances of the body of the `whenever`.

Examples. Fig. 6 illustrates the occurrences of `Q` on a curve sampled every 10^{-2} seconds:

```
@pattern Q {
  @local $s1, $t1, $s2, $t2
  @refractory 2

  State $X start $s1 stop $t1 during[0.5]
    where $X > [1#]:$X      // Q1
  Before[1.3]
    State $X start $s2 stop $t2 during[0.5]
    where $X > [1#]:$X      // Q2
}
```

This pattern matches two intervals, of length 0.5, separated by less than 1.3 time units, such that on this interval `$X > [1#]:$X` holds. The notation `[1#]:$X` is used to access the past value of `$X` (the specification `[n#]` corresponds to the value at the n to the last update. In other word, this property simply characterizes a series of increasing values.

The input signal plotted on Fig. 6, is increasing on $[0, 1]$, $[2, 3]$ and $[4, 4.6]$. It is decreasing on $[1, 2]$ and $[3, 4]$. There are two occurrences of `Q`: the first matches the intervals $[0.49, 0.99]$, $[2.05, 2.55]$ and the second the intervals $[2.48, 2.98]$, $[4.08, 4.58]$. Notice that the second match cannot ends in the interval $[2.55, 2.55 + 2]$ because of the refractory period. The small shift in the interval boundaries is caused by a small lag phase in the sampling (so, there is no sampling point at integer time coordinates).

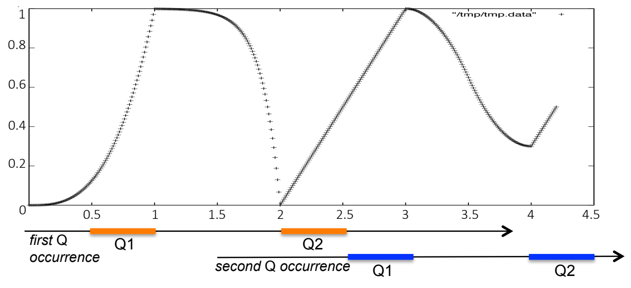


Figure 6. The plot shows the signal on which the pattern `Q` is matched. The events correspond to the sampling of an arbitrary curve every $1/100$s. The occurrences of `Q` are outline on the time axes where the matching of the `State` sub-patterns `Q1` and `Q2` have been outlined.

Note that after the match of the first `State` pattern `Q1`, each update of the variable `$X` every 10^{-2}s, a `whenever` is triggered to look for the second `State` pattern `Q2`. So, at the end of the temporal scope of 1.3, they can be about 50 active parallel `whenever` for `Q2` only (because of the effect of the during clause). This is not a problem for the current implementation, even if we lower the sampling rate by a factor of 4. We are not able to push the system further because the *Antescofo* system is embedded in the Max/MSP environment which allows a time slot for *Antescofo* computation only every 2ms.

7. Conclusions

Research around the *Antescofo* system focuses on how to achieve a high-level musical interaction between live musicians and a computer. Temporal patterns extend the *Antescofo* domain specific language for the out-of-time specification of complex timed sequences of events.

We presented only a subset of the available constructs. In particular, we have not discussed the `NoEvent` atomic pattern that can be used to check the absence of an event of a given kind over a definite period. But the fragment presented here is sufficient enough to give a flavor of the temporal constructions, especially pattern variables, the possibility to deal with temporal bounds in term of number of logical events as well as in term of metric time, the distinction between properties satisfied on an event or on an interval, the constraints brought by the online evaluation and the causality, the earliest match property and the notion of refractory period.

The semantics developed here do not face the problem of being integrated in the semantics of the entire *Antescofo* DSL. A semantic for the static kernel of the DSL has been given in term of timed-automata in [13]. For the sake of the simplicity, we have defined the matching function on a given time-event sequence whilst the actions triggered by the occurrence of pattern may generate new events. But the handling of such recursion is orthogonal to the problem of defining the meaning of temporal patterns.

We plan to continue our research in several directions. First, we will explore issues related to hierarchy and grouping. We will also extend the pattern language, *e.g.* to include pattern matching on *Antescofo* data structure, following the approach of [16, 17] and to support uncertainty. Second, it will be useful to investigate alternative implementations. For instance, using the history mechanism on variables, it is possible to implement the example of Fig. 3 with only one `whenever`. How histories may simplify the handling of `State` pattern and metric `Before` is much less clear. Finally, we will study the applicability of temporal patterns to the implementation of audio processing, especially for spectral computations. This kind of computations is even more computationally demanding and requires a better handling of time and space resources.

8. Acknowledgments

The authors wish to thank Arshia Cont, the members of the MuTAnt project, Hanna Klaudel, Antoine Spicher and the members of the SynBioTIC group for their support and comments. Discussions with the composers Julia Blondeau, Gilbert Nouno and José Miguel Fernandez were always constructive and fruitful, providing bold motivations and strong applications. This work has been partly supported by the ANR projects SymBioTIC (ANR-10-BLAN-0307) and INEDIT (ANR-12-CORD-0009).

References

[1] A. Alfonso, V. Braberman, N. Kicillof, and A. Olivero. Visual timed event scenarios. In *Proceedings of the 26th International Conference on Software Engineering*, pages 168–177. IEEE Computer Society, 2004.

[2] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM (JACM)*, 41(1):181–203, 1994.

[3] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *Journal of the ACM*, 49(2):172–206, 2002.

[4] A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 20(4):14, 2011.

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[6] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2):87–152, 1992.

[7] P. Bouyer, F. Chevalier, and D. D'Souza. Fault diagnosis using timed automata. In *Foundations of Software Science and Computational Structures*, pages 219–233. Springer, 2005.

[8] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[9] A. Cont. *Antescofo*: Anticipatory synchronization and control of interactive parameters in computer music. In *Proceedings of International Computer Music Conference (ICMC)*. Belfast, August 2008.

[10] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, 44(3):15, 2012.

[11] R. B. Dannenberg. An on-line algorithm for real-time accompaniment. In *Proceedings of the International Computer Music Conference (ICMC)*, pages 193–198, 1984.

[12] D. D'souza. A logical characterisation of event clock automata. *Int. J. of Foundations of Computer Science*, 14(04):625–639, 2003.

[13] J. Echeveste, A. Cont, J.-L. Giavitto, and F. Jacquemard. Operational semantics of a domain specific language for real time musician–computer interaction. *Discrete Event Dynamic Systems*, 23(4):343–383, 2013.

[14] C. Elliott and P. Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[15] J. Freeman. Web-based collaboration, live musical performance and open-form scores. *International Journal of Performance Arts and Digital Media*, 6(2):149–170, 2010.

[16] J.-L. Giavitto and O. Michel. Declarative definition of group indexed data structures and approximation of their domains. In *Proceedings of the 3rd ACM PPDP Conference*, pages 150–161. ACM, 2001.

[17] J.-L. Giavitto, O. Michel, and J. Cohen. Pattern-matching and rewriting rules for group indexed data structures. *SIGPLAN Not.*, 37(12):76–87, Dec. 2002. Selected papers from the workshops of the fourth Principles, Logics, and Implementations of High-Level Programming Languages (PLI) confederation of conferences.

[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[19] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-time systems*, 2(4):255–299, 1990.

[20] E. A. Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, 2009.

[21] D. C. Luckham. *The power of events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002.

[22] L. Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46. ACM, 2008.

[23] F. Maraninchi and Y. Rémond. Mode-automata: a new domain-specific construct for the development of safe critical systems. *Science of Computer Programming*, 46(3):219–254, 2003.

[24] P. D. Mosses. *Handbook of Theoretical Computer Science*, volume 2, chapter Denotational Semantics, pages 575–631. Elsevier Science, 1990.

[25] N. W. Paton and O. Díaz. Active database systems. *ACM Computing Surveys (CSUR)*, 31(1):63–103, 1999.

[26] M. Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77, 1991.

[27] M. Roesch et al. Snort: Lightweight intrusion detection for networks. In *LISA*, volume 99, pages 229–238, 1999.

[28] M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. In *Computer Security Foundations Workshop, IEEE*, pages 0220–0220. IEEE Computer Society, 2001.

[29] M. H. Smith, G. J. Holzmann, and K. Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 14–22. IEEE, 2001.

[30] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for concurrency*, pages 238–266. Springer, 1996.

[31] B. Vercoe. The synthetic performer in the context of live performance. In *Proceedings of the ICMC*, pages 199–200, 1984.

[32] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2006.