Monitoring Dense-Time, Continuous-Semantics, Metric Temporal Logic

Kevin Baldor[1,2] and Jianwei Niu[1]

[1] University of Texas at San Antonio, USA
{kbaldor,niu}@cs.utsa.edu
[2] Southwest Research Institute, San Antonio, USA

Abstract. The continuous semantics and dense time model most closely model the intuitive meaning of properties specified in metric temporal logic (MTL). To date, monitoring algorithms for MTL with dense time and continuous semantics lacked the simplicity the standard algorithms for discrete time and pointwise semantics. In this paper, we present a novel, transition-based, representation of dense-time boolean signals that lends itself to the construction of efficient monitors for safety properties defined in metric temporal logic with continuous semantics. Using this representation, we present a simple lookup-table-based algorithm for monitoring formulas consisting of arbitrarily nested MTL operators. We examine computational and space complexity of this monitoring algorithm for the past-only, restricted-future, and unrestricted-future temporal operators.

1 Introduction

Program monitoring has attracted interest as an alternative to model checking or theorem proving as these become impractical due to the size of the state space of a full program. A dynamic analysis, monitoring is limited to the detection of property violations that are actually observed in a program's execution, and more fundamentally, to so called *safety properties*, those that can be falsified given only a finite number of program events.

Temporal logics such as linear temporal logic (LTL) [7] and computation tree logic (CTL) [3] provide an effective formal description for desired or undesired program behavior and are commonly used in monitoring applications. Each of these logics specify constraints on the order of occurrence of events. For example, they can state that "after event p, event q must take place at some point in the future". This is not an enforceable safety property but would become one if modified to state that q must occur within a certain period of time. To do so, these logics must be augmented with an explicit notion of time.

One such augmentation is metric temporal logic (MTL) [5]. It introduces limits on the periods of time over which a logical connective operates. For example, the property described in the preceding paragraph may be specified as $p \rightarrow \Diamond_{[0,5]}\, q$. The subscript on the *eventually* operator ($\Diamond$), is an interval – relative to the current time – in which q must hold in order for the statement to be true at the current time. Some notations support a number of subscript forms, but without loss of generality, we restrict our presentation to the use of intervals that may be closed or open on either end. Additionally, we admit intervals of the form $[a, a]$, though their use incurs a potential space penalty.

The runtime-verification community employs two time models for MTL: discrete and dense. Within the dense-time model, there are two semantics: point-based and continuous [8] [2]. We concentrate on the latter, as in [2], Basin et al. assert that "Real-time logics based on a dense, interval-based time model are more natural and general than their counterparts based on a discrete or point-based model". But in it, they present a monitoring algorithm that they describe as "conceptually simpler" for the point-based semantics than for the interval-based (continuous) semantics.

Our contribution with this paper is the introduction of a transition-based – rather than interval-based [2] – representation for the dense-time boolean signals that are a feature of the continuous semantics. With this representation, the output of all MTL connectives can be expressed as a simple lookup-table indexed by the input. Using this representation, we present a conceptually simple MTL monitoring algorithm modeled on the transducer-approach of [6] that reduces to something like the LTL-monitoring algorithm of [4] for past-only operators. We then observe the increase in space complexity of its extension to future MTL expressions.

2 Background

2.1 LTL and MTL

Logical expressions use the connectives for disjunction ($\vee$), *logical or*; conjunction ($\wedge$), *logical and*; and negation ($\neg$) to describe the relationship between logical statements at the current time. Linear Temporal Logic (LTL) augments them with a number of logical connectives that describe the relationship between logical expressions over time.

The past-only operator *historically* ($\boxminus\phi$) indicates that the expression ϕ has been true since time zero, *once* ($\diamondminus\phi$) indicates that ϕ must have been true at some point in time since time zero, and *since* ($\phi\ \mathcal{S}\ \psi$) indicates that at some point in the past ψ must have been true and that ϕ must at least have been true at every point after that until the current time;

The future-only operator *globally* ($\Box\phi$) indicates that the expression ϕ is true now and will be true at all points in the future, *eventually* ($\Diamond\phi$) indicates that ϕ must be true at the current time or at some point in the future, and *until* ($\phi\,\mathcal{U}\ \psi$) indicates that at the current time or at some point in the future ψ must be true and that that ϕ must at least have been true at every point between the current time and that point.

The semantics of LTL operate on a trace, a countably infinite sequence of truth values of atomic elements. LTL expressions are only interpreted as having a truth value at the instants of time corresponding to the elements of the trace. Beyond the order of events, the actual time at which the events plays no role in determining the truth of the LTL expressions.

The pointwise semantics of MTL are a natural extension of the semantics of LTL in that, while they augment the order constraints of LTL with true time constraints, the truth of an expression is only defined at discrete points in time. When used for monitoring, an MTL expression might be evaluated only when an input event arrives. This is more efficient than periodically re-evaluating expressions, but can lead to counter-intuitive results. In [2], Basin et al. discuss a number of such results. Perhaps most striking is that under pointwise semantics $\Diamond_{[0,1]}\Diamond_{[0,1]}\phi$ is not logically equivalent to $\Diamond_{[0,2]}\phi$. This is illustrated in the case that ϕ is true at time $\tau = 0$, and the next observation of the system takes place at time $\tau = 2$, $\Diamond_{[0,2]}\phi$ is *true* at time $\tau = 2$, but $\Diamond_{[0,1]}\Diamond_{[0,1]}\phi$ is *false* since the observations lack the 'bridge' at time $\tau = 1$ (for which $\Diamond_{[0,1]}\phi$ would evaluate to *true*) needed to declare $\Diamond_{[0,1]}\Diamond_{[0,1]}\phi$ to be *true* at time $\tau = 2$.

As observed by the authors of [2], adding additional sample points can restore the equivalence of these expressions at the cost of additional computation by the monitor. In a discrete-valued-time system, this can be taken to the extreme of evaluating all expressions with each 'clock tick'. Beyond the computational cost, this cannot be extended to the dense-time representation that best models external events for which there is no shared clock.

2.2 Continuous Semantics and Boolean Signals

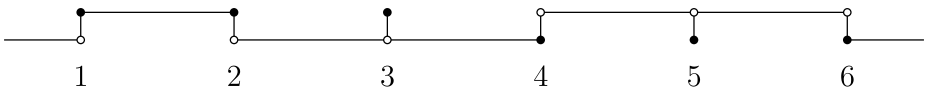
Under continuous semantics, we avoid the ambiguity introduced by the selection of sample points. Loosely, the continuous semantics (MTL) assign a truth value to any expression for any point of time greater than zero. A more formal definition is presented in [2], but we will present enough here for the purpose of discussion. Essentially, the notion of a trace used in LTL and the pointwise semantics of MTL is replaced by a mapping from time $\tau \in \mathbb{R}_{\geq 0}$ to $\{$*true*, *false*$\}$ that the authors term a boolean signal. In their formulation, the boolean signal for the expression ϕ, denoted γ_ϕ, is the set of all points in time for which ϕ evaluates to true. In Figure 1 – expanded from the definitions given in [2] to include $\mathcal{U}_I$, the set of all signals in a model is written $\hat{\gamma}$, $\tau \in \mathbb{R}_{\geq 0}$ denotes the time for which the statement applies, and the subscript I is an interval on $\mathbb{R}$ for which the operator applies.

$$
\begin{array}{lll}
\hat{\gamma}, \tau \models p & \text{iff} & \tau \in \gamma_p \\
\hat{\gamma}, \tau \models \neg\phi & \text{iff} & \hat{\gamma}, \tau \not\models \phi \\
\hat{\gamma}, \tau \models \phi \wedge \psi & \text{iff} & \hat{\gamma}, \tau \models \phi \text{ and } \hat{\gamma}, \tau \models \psi \\
\hat{\gamma}, \tau \models \phi\ \mathcal{S}_I\ \psi & \text{iff} & \exists\ \tau' \in [0,\tau] \text{ such that } \tau - \tau' \in I,\ \hat{\gamma}, \tau' \models \psi, \text{ and } \hat{\gamma}, \kappa \models \phi\ \forall \kappa \in (\tau', \tau] \text{ or}[1] \\
 & & \exists\ \tau'' \in [0,\tau') \text{ such that } \tau - \tau'' \in I,\ \hat{\gamma}, \kappa \models \psi\ \forall \kappa \in [\tau'', \tau') \text{ and } \hat{\gamma}, \kappa \models \phi\ \forall \kappa \in [\tau', \tau] \\
\hat{\gamma}, \tau \models \phi\ \mathcal{U}_I[2]\ \psi & \text{iff} & \exists\ \tau' \geq \tau \text{ such that } \tau' - \tau \in I,\ \hat{\gamma}, \tau' \models \psi, \text{ and } \hat{\gamma}, \kappa \models \phi\ \forall \kappa \in [\tau, \tau') \text{ or} \\
 & & \exists\ \tau'' > \tau' \text{ such that } \tau'' - \tau \in I,\ \hat{\gamma}, \kappa \models \psi\ \forall \kappa \in (\tau', \tau''] \text{ and } \hat{\gamma}, \kappa \models \phi\ \forall \kappa \in [\tau, \tau']
\end{array}
$$

Fig. 1. Continuous Semantics of MTL

3 Modeling Dense-Time Boolean Signals as Event Sequences

The dense model of time precludes a representation consisting of every value for which a boolean signal is true as there may be uncountably many such points. However, boolean signals are defined as satisfying the finite-variability condition that on any bounded interval there exists a finite number of non-overlapping intervals over which the signal is *true*. This lends itself to the representation used in [2] an at-most-countably-infinite set of non-overlapping intervals. Our representation differs in that we model boolean signals not as a series of intervals, but as a sequence of timed events denoting a transition from one truth value to another. For example, a signal described with the intervals $\{[1,2], [3,3], (4,5), (5,6)\}$ might be illustrated as



where the higher line indicates *true* and the lower, *false*. The dots at the transition indicate whether the signal is considered *true* at the transition point. We represent this signal as the series of transitions

$$\{(\overset{\bullet}{\uparrow}, 1), (\overset{\bullet}{\downarrow}, 2)\ (\overset{\bullet}{\updownarrow}, 3)\ (\underset{\bullet}{\uparrow}, 4), (\underset{\bullet}{\updownarrow}, 5)\ (\underset{\bullet}{\downarrow}, 6)\}$$

This example exhausts all of the transition types required to describe boolean signals. In the following sections the additional events $(\_, \tau)$ and $(^{-}, \tau)$ are used to indicate the lack of transition on one of the inputs of a binary operator. They are not strictly required to unambiguously describe a boolean signal, but are convenient for the implementation of the monitor.

[1] This clause was added to capture another boundary condition introduced by the dense time model.

[2] We introduce the *until* operator using the presentation of [2] to relate the definition of future operators in the standard definitions of the runtime-verification community.

Definition 1. *A boolean signal is described by an infinite sequence of timed transitions* (δ_i, τ_i) *for which* $\delta_0 \in \{\_, \overline{\ }, \uparrow, \downarrow\}$, $\delta_i \in \{\uparrow, \uparrow, \bot, \downarrow, \downarrow, \top\} \forall\ i > 0$, *and* $\tau_i \in \mathbb{R}_{\geq 0}$, $\tau_0 = 0$, *and* $\tau_{i+1} > \tau_i \forall i \geq 0$. *The timed transitions are subject to the further constraint types of adjacent transitions must agree in the sense that the incoming value of each transition must match the outgoing value of the previous transition. More formally,*

$$\delta_i \in \{\overline{\ }, \uparrow, \uparrow, \top\} \rightarrow \delta_{i+1} \in \{\downarrow, \downarrow, \top\} \textit{ and}$$
$$\delta_i \in \{\_, \downarrow, \downarrow, \bot\} \rightarrow \delta_{i+1} \in \{\uparrow, \uparrow, \bot\}.$$

Definition 2. *The truth of a signal* γ *at time* τ *is given by*

$$\tau \in \gamma \doteq \begin{cases} \delta_k \in \{\overline{\ }, \uparrow, \downarrow, \bot\} & \exists k : \tau_k = \tau \\ \delta_k \in \{\overline{\ }, \uparrow, \uparrow, \bot\} & \exists k : \tau_k < \tau \wedge \ (\tau_{k+1} > \tau \vee k = |\gamma|) \end{cases}$$

4 Monitor Construction

4.1 Supported Temporal Operators

Although the timed *until* and *since* are sufficient to capture MTL semantics, the treatment of their transitions is sufficiently complicated that we follow the approach of [6] and introduce timed *eventually* to enable the treatment of only the non-metric *until* and *since*. This is accomplished by exploiting the fact that timed *since* and *until* are redundant given timed *historically* ($\boxminus_I$) , *once* ($\diamondminus_I$), *henceforth* ($\Box_I$), and *future* ($\Diamond_I$) since

$$\phi\ \mathcal{S}_{[a,b]}\psi \leftrightarrow \boxminus_{[0,a]}(\phi\ \mathcal{S}\ \psi) \wedge\ \diamondminus_{[a,b]}\psi \text{ and}$$
$$\phi\ \mathcal{U}_{[a,b]}\psi \leftrightarrow \Box_{[0,a]}(\phi\ \mathcal{U}\ \psi) \wedge\ \Diamond_{[a,b]}\psi$$

and that $\boxminus_I$ and $\Box_I$ are redundant since

$$\boxminus_I\phi \leftrightarrow \neg\diamondminus_I\neg\phi$$
$$\Box_I\phi \leftrightarrow \neg\Diamond_I\neg\phi$$

Further, we observe that for the same input the output of $\Diamond_{[a,a+\varDelta]}$ and $\diamondminus_{[b,b+\varDelta]}$ are both simply a time-shifted version of the output of $\Diamond_{[0,\varDelta]}$. By generalizing the intervals to support negative indices, we obtain

$$\diamondminus_{[a,b]}\phi \leftrightarrow \Diamond_{[-b,-a]}\phi$$

As a result, we can monitor both future and past MTL using only the transducers for the operators $\neg$, $\wedge$, $\mathcal{S}$, $\mathcal{U}$, and $\Diamond_I$, the formal definitions for which are given in Figure 2.

$$\begin{array}{lll}
\hat{\gamma},\tau \models \neg\phi & \text{iff} & \hat{\gamma},\tau \not\models \phi \\
\hat{\gamma},\tau \models \phi \wedge \psi & \text{iff} & \hat{\gamma},\tau \models \phi \text{ and } \hat{\gamma},\tau \models \psi \\
\hat{\gamma},\tau \models \Diamond_I\ \phi & \text{iff} & \exists\ \tau' \text{such that } \tau' - \tau \in\ I,\ \hat{\gamma},\tau' \models\ \phi \\
\hat{\gamma},\tau \models \phi\ \mathcal{S}\ \psi & \text{iff} & \exists\ \tau' \in\ [0,\tau] \text{ such that } \hat{\gamma},\tau' \models \psi \text{ and } \hat{\gamma},\kappa \models \phi\ \forall\kappa \in (\tau',\tau] \text{ or} \\
 & & \quad \exists\ \tau'' \in\ [0,\tau') \text{ such that } \hat{\gamma},\kappa \models \psi\ \forall\kappa \in [\tau'',\tau') \text{ and } \hat{\gamma},\kappa \models \phi\ \forall\kappa \in [\tau',\tau] \\
\hat{\gamma},\tau \models \phi\ \mathcal{U}\ \psi & \text{iff} & \exists\ \tau' \geq \tau \text{ such that } \hat{\gamma},\tau' \models \psi \text{ and} \hat{\gamma},\kappa \models \phi\ \forall\kappa \in (\tau',\tau] \text{ or} \\
 & & \quad \exists\ \tau'' > \tau' \text{ such that } \hat{\gamma},\kappa \models \psi\ \forall\kappa \in (\tau',\tau''] \text{ and } \hat{\gamma},\kappa \models \phi\ \forall\kappa \in [\tau,\tau']
\end{array}$$

Fig. 2. Semantics of Monitored MTL Connectives

4.2 Monitoring Algorithm

To monitor a formula ϕ, we begin by converting it into a parse tree. From this, we construct an array Φ consisting of one transducer for each node of the parse tree in reverse-topological-sort order. That is, for any node in the parse tree, its children appear before it in Φ. The transducers maintain some operation-specific fields, but each contains at least $\langle op, inputs, Q, I_{\text{valid}}\rangle$ where *op* identifies the operation, *inputs* contains a pointer to the elements of Φ upon which it depends, Q is a queue containing the output of the transducer, and I_{valid} indicates the time interval over which the output of the transducer is valid.

The valid interval allows a transducer to 'stop time' while its state is undetermined. For example, the transducer for *eventually* uses this to apply a constant offset to all output transitions, whereas the *until* transducer may delay its output for an indeterminate period. Even the simple transducers such as negation and conjunction must be able to specify a valid interval since their input might do so.

Some transducers define additional state variables in addition to those mentioned above. The *since* transducer maintains a state indicating whether or not the latest transition left its output in the UP state; The *eventually* transducer maintains a timer that is used by the monitoring algorithm to call UPDATE again at some point in the future.

In the following pseudocode, the UPDATE procedures for each transducer employs transition tables such as $\text{FUTURE}_\text{a}[\delta]$ for the *eventually* transducer. Their contents are given in the following sections.

The monitoring procedure consists of gathering an ensemble of simultaneous transition events for the external inputs to the monitor and storing them in a container, Δ, that maps input variable names to transition types. If a timer expires, UPDATE may be called with no input transitions. The UPDATE function returns the next timer expiration time so that the monitor may call it when it has expired. The following pseudocode describes the most general version of the update operation and the UPDATE procedures for the more interesting transducers[3]. Subsequent sections will describe the simplifications that are possible when supporting subsets of MTL.

```
function UPDATE($\Phi$, $\Delta$, $\tau$)
    for $\varphi \in \Phi$ do
        if $\varphi$.op $\in$ input variables then
            if $\varphi$.op.id $\in \Delta$ then
                ENQUEUE($\varphi_Q$,$\Delta[\varphi$.op.id], $\tau$)
                $\varphi.I_{\text{valid}} \leftarrow [0, \tau]$
        else
            while $(\delta^*, \tau) \leftarrow$ SYNC($\varphi$.inputs) do
                UPDATE$_{\varphi.\text{op}}(\varphi, \delta^*, \tau)$
        UPDATEVALIDINTERVAL$_{\varphi.\text{op}}(\varphi)$
    return min({$\varphi.\tau_{\text{timer}}$ for $\varphi \in \Phi$})
```

```
function UPDATE$_{\Diamond_I}(\varphi, \delta, \tau)$
    $\tau' \leftarrow \tau - b$                    ▷ output time offset
    if $\tau = 0$ then
        ENQUEUE($\varphi.Q$, FUTURE$_{\text{INIT}}[\delta], \tau'$)
    else if $\tau = \varphi.\tau_{\text{timer}}$ then
        if $\delta = \varnothing$ then
            ENQUEUE($\varphi.Q$, $\varphi.\delta_{\downarrow}, \tau'$)
        else if $\varphi.\delta_{\downarrow} =$ ↧ then
            ENQUEUE($\varphi.Q$, FUTURE$_{\text{c}}[\delta], \tau'$)
        $\varphi.\delta_{\downarrow} \leftarrow \varnothing$
        $\varphi.\tau_{\text{timer}} \leftarrow \varnothing$
    else if $\delta \in$ FUTURE$_{\text{a}}$ then
        ENQUEUE($\varphi.Q$, FUTURE$_{\text{a}}[\delta], \tau'$)
    if $\delta \in$ FUTURE$_{\text{b}}$ then          ▷ down transition
        $\varphi.\delta_{\downarrow} \leftarrow$ FUTURE$_{\text{b}}[\delta]$
        $\varphi.\tau_{\text{timer}} \leftarrow \tau + b - a$
```

```
function UPDATEVALIDINTERVAL$_{\Diamond_I}(\varphi)$
    $\phi \leftarrow \varphi$.inputs
    switch $\phi.I_{\text{valid}}$
        case $[0, i]$    $\varphi.I_{\text{valid}} \leftarrow [0, i - b]$
        case $[0, i)$    $\varphi.I_{\text{valid}} \leftarrow [0, i - b)$
```

```
function UPDATE$_{\mathcal{U}}(\varphi, \delta_\phi, \delta_\psi, \tau)$
    if $\tau = 0$ then
        ENQUEUE($\varphi.Q$, UNTIL$_{\text{INIT}_\text{a}}[\delta_\phi, \delta_\psi], \tau$)
        $\varphi.\delta_{\uparrow} \leftarrow$ UNTIL$_{\text{INIT}_\text{b}}[\delta_\phi, \delta_\psi]$
        $\varphi.\delta_{\downarrow} \leftarrow$ UNTIL$_{\text{INIT}_\text{c}}[\delta_\phi, \delta_\psi]$
        $\varphi.\tau_{\text{pending}} \leftarrow \tau$
    else
        switch UNTIL$_{\text{a}}[\delta_\phi, \delta_\psi]$
            $\tau' \leftarrow \varphi.\tau_{\text{pending}}$
            case ↥
                ENQUEUE($\varphi.Q$, $\varphi.\delta_{\uparrow}$, $\tau'$)
            case ↧
                ENQUEUE($\varphi.Q$, $\varphi.\delta_{\downarrow}$, $\tau'$)
        ENQUEUE($\varphi.Q$, UNTIL$_{\text{b}}[\delta_\phi, \delta_\psi], \tau$)
        $\varphi.\delta_{\uparrow} \leftarrow$ UNTIL$_{\text{c}}[\delta_\phi, \delta_\psi]$
        $\varphi.\delta_{\downarrow} \leftarrow$ UNTIL$_{\text{INIT}_\text{d}}[\delta_\phi, \delta_\psi]$
        $\varphi.\tau_{\text{pending}} \leftarrow \tau$
    if $\neg(\varphi.\delta_{\uparrow} = \varphi.\delta_{\downarrow} = \varnothing)$ then
        $\varphi.I_{\text{valid}} \leftarrow [0, \tau)$
```

```
function UPDATEVALIDINTERVAL$_{\mathcal{U}}(\varphi)$
    $\phi, \psi \leftarrow \varphi$.inputs
    if $\varphi.\delta_{\uparrow} = \varphi.\delta_{\downarrow} = \varnothing$ then
        $\varphi.I_{\text{valid}} \leftarrow \phi.I_{\text{valid}} \cap \psi.I_{\text{valid}}$
```

[3] For the complete pseudocode, see the full version of this paper [1]

```
function Enqueue($Q$, $\delta$, $\tau$)
    if $\delta = \varnothing$ then return
    case $\tau < 0$
        Clear($Q$)
        if $\delta \in \{↑, ⇡, ⊤\}$ then
            Append($Q$,(‾,0))
        else
            Append($Q$(_,0))
    case $\tau = 0$
        Clear($Q$)
        case $\delta = ↑$
            Append($Q$(‾,0))
        case $\delta \in \{⊤, ⇡\}$
            Append($Q$(⇡,0))
        case $\delta \in \{⊥, ↓\}$
            Append($Q$(↓,0))
        case $\delta = ⇣$
            Append($Q$(_,0))
    case $\tau > 0$
        if Empty($Q$) $\wedge$ $Q$.value $= \varnothing$ then
            if $\delta \in \{$_, ‾$\}$ then
                $Q$.append(($\delta$, 0))
                return
            else if $\delta \in \{↑, ⇡, ⊤\}$ then
                $Q$.append((_, 0))
            else
                $Q$.append((‾, 0))
        else
            $Q$.append(($\delta$, $\tau$))
```

```
function Dequeue($Q$)
    ($\delta$, $\tau$) $\leftarrow$ RemoveFirst($Q$)
    case $\delta \in \{$‾, ↑, ⇡, ⊤$\}$
        $Q$.value $\leftarrow$ ‾
    case $\delta \in \{$_, ⇣, ↓, ⊥$\}$
        $Q$.value $\leftarrow$ _
    return ($\delta$, $\tau$)
```

```
function Sync(inputs)
    $\varphi, \psi \leftarrow$ inputs
    if $\psi = \varnothing$ then                                ▷ one input
        if Empty($\varphi.Q$) then
            return $\varnothing$
        return Dequeue($\varphi.Q$)
    else                                                          ▷ two inputs
        $I_{\text{valid}} \leftarrow \varphi.I_{\text{valid}} \cap \psi.I_{\text{valid}}$
        $e_\varphi \leftarrow$ Head($\varphi.Q$) if Head($\varphi.Q$).$\tau \in I_{\text{valid}}$
        $e_\psi \leftarrow$ Head($\psi.Q$) if Head($\psi.Q$).$\tau \in I_{\text{valid}}$
        if $e_\varphi \neq \varnothing \wedge e_\psi \neq \varnothing$ then
            if $e_\varphi.\tau = e_\psi.\tau$ then
                Dequeue($\varphi.Q$)
                Dequeue($\psi.Q$)
                return $((e_\varphi.\delta, e_\psi.\delta), e_\varphi.\tau)$
            if $e_\varphi.\tau < e_\psi.\tau$ then
                $e_\psi \leftarrow \varnothing$
            else
                $e_\varphi \leftarrow \varnothing$
        if $e_\varphi \neq \varnothing$ then
            Dequeue($\varphi.Q$)
            return $((e_\varphi.\delta, \psi.Q.value), e_\varphi.\tau)$
        else
            Dequeue($\psi.Q$)
            return $((\varphi.Q.value, e_\psi.\delta), e_\varphi.\tau)$
```

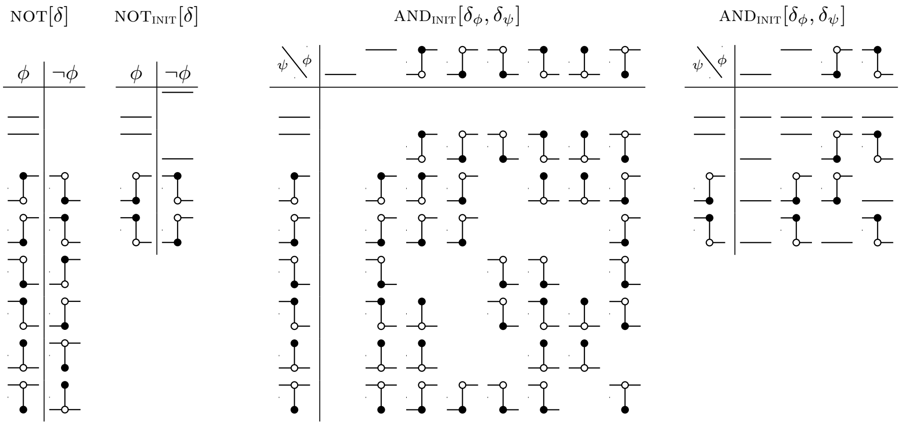
Missing from this elided version of the pseudocode are the Update procedures for negation, conjunction, and the *since* operator. They are simpler than $\Diamond_I$ and $\mathcal{U}$ and the general sense of their operation is given in the sections describing their transducer tables.

The functions on the second page support the Update procedures and simplify their logic. For example, in addition to its obvious purpose, Enqueue ensures that the outputs obey the transition rules described in definition 1. Dequeue ensures that the the value of a boolean signal can be obtained at times for which there is no transition event per definition 2. Using this behavior of the Dequeue procedure, the Sync procedure produces from two Boolean signals an ordered stream of events for all time points at which either signal exhibits a transition event. This is needed to drive the transducer tables for those operators that have two inputs.

5 Signal Transducer Tables

5.1 Negation and Conjunction

Figure 3 contains the transition tables for negation and conjunction. The INIT versions are used for the first transition and the regular version for all subsequent transitions. The non-transition states of the inputs are illustrated, but table entries for which there is no transition are omitted in the interest of readability. Note that for the INIT versions of the tables, the non-transition outputs (‾ and _) are shown because they are actually produced for the initial entry of a boolean signal.



NOT[δ]
NOT$_{\text{INIT}}$[δ]
AND$_{\text{INIT}}$[δ_ϕ, δ_ψ]
AND$_{\text{INIT}}$[δ_ϕ, δ_ψ]
ϕ
$\neg\phi$
ψ
ϕ

Fig. 3. Transition Tables for the Negation and Conjunction Operators

5.2 Since

The transducer for $\phi\ \mathcal{S}\ \psi$ is somewhat more complicated. The output transition for a given set of input transitions is influenced by whether or not the since operator is currently in the down or the up state. As in the previous section, omitted entries indicate states for which there is no transition. Some of the omitted entries indicate states that cannot be reached, but these are not specially indicated – nor is any special handling required to deal with unreachable states.

The transducer begins by applying the table SINCE$_{\text{INIT}}[\delta_\phi, \delta_\psi]$ for the initial transition, afterward the tables SINCE$_{\text{UP}}[\delta_\phi, \delta_\psi]$ and SINCE$_{\text{DOWN}}[\delta_\phi, \delta_\psi]$ are used. Which of the tables is to be used is determined by the type of transition last emitted. If it is in {‾, ⌠, ⌠, ⊤}, then the UP table is used, otherwise, it is the DOWN table.

5.3 Eventually

The metric eventually operator uses the tables from Figure 5. Its transducer is distinguished from those introduced thus far by the addition of a timer used

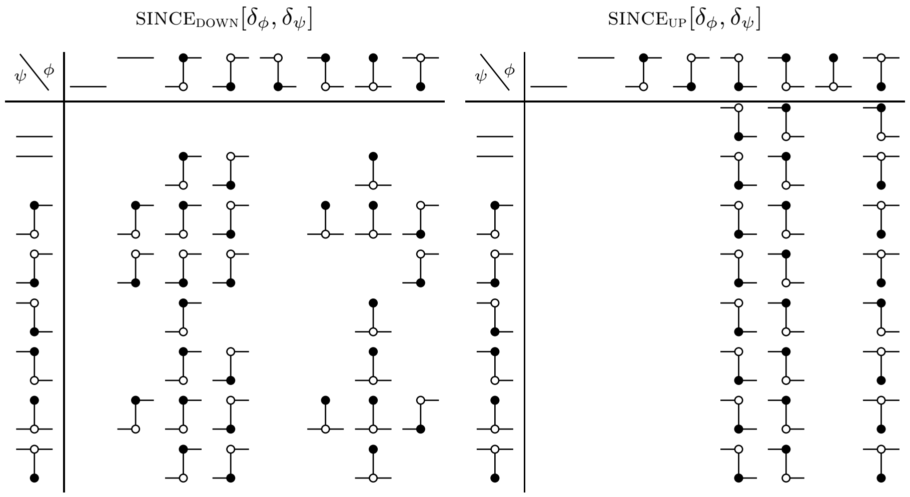

SINCE_DOWN[δφ, δψ]
SINCE_UP[δφ, δψ]
ψ φ
ψ φ

Fig. 4. Transition Tables for the *Since* Operator, $\phi\ \mathcal{S}\ \psi$

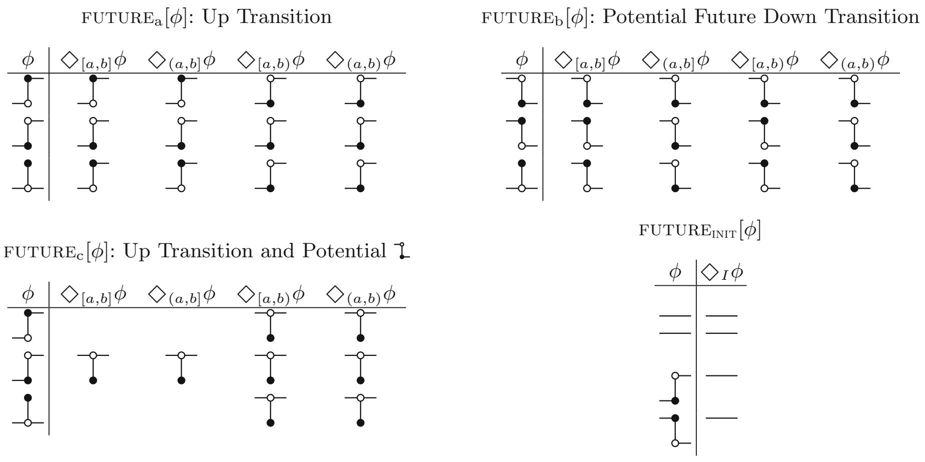

FUTURE_a[φ]: Up Transition
FUTURE_b[φ]: Potential Future Down Transition
FUTURE_c[φ]: Up Transition and Potential
FUTURE_INIT[φ]
φ
◇[a,b]φ
◇(a,b]φ
◇[a,b)φ
◇(a,b)φ
◇Iφ

Fig. 5. Transition Tables for the *Eventually* Operator

to generate a down event $b - a$ time units after the input transitions to the down state. All events emitted in response to an event at time τ are emitted at time $\tau - b$. Also, it can enter an indeterminate state in response to a down transition. When a down transition occurs, the transducer stores the potential down transition – the type of which is determined by the interval type – in the state variable $\delta_{\downarrow}$. The actual transition emitted can be affected if an up transition occurs before or simultaneous with the timer expiration.

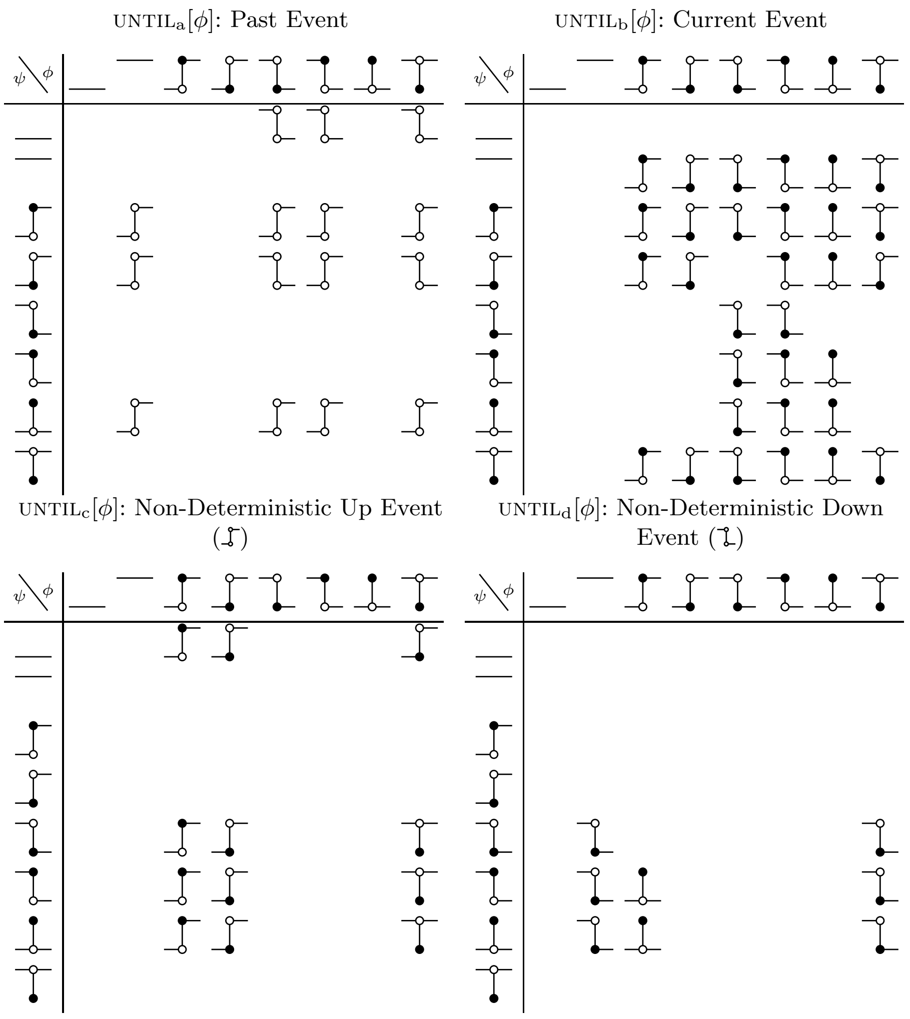

UNTILa[ϕ]: Past Event
UNTILb[ϕ]: Current Event
UNTILc[ϕ]: Non-Deterministic Up Event
UNTILd[ϕ]: Non-Deterministic Down Event
ψ
ϕ

Fig. 6. Transition Tables for the *Until* Operator, $\phi\,\mathcal{U}\,\psi$

5.4 Until

The transducer for the *until* operator can enter a more complicated indeterminate state than that of the non-metric *eventually* operator; It can maintain a different potential transition depending on whether it is ultimately found to be *true* or *false* at the point in time at which its output became uncertain. The simplest example occurs when monitoring $\phi\,\mathcal{U}\,\psi$ and ϕ becomes *true* with transition type ⟟ at a time τ when ψ is *false*. If ψ becomes *true* before ϕ becomes *false*, then the transducer should emit the transition event (⟟, τ); If ϕ becomes *true* before ψ becomes *false*, this potential transition is abandoned and the output remains *false* up to and including the current time. It is possible that the transducer will maintain two such potential transitions, $\delta_{\downarrow}$ or $\delta_{\uparrow}$. We introduce the notations *non-deterministic up* $\uparrow$ and *non-deterministic down* $\downarrow$ in UNTIL$_{\mathrm{a}}$ to denote which of the potential transitions is to be emitted upon the arrival of input events.

6 Correctness

Theorem 1. [4] *The* UPDATE *procedure applied to the above transducer tables correctly models the semantics of Figure 1.*

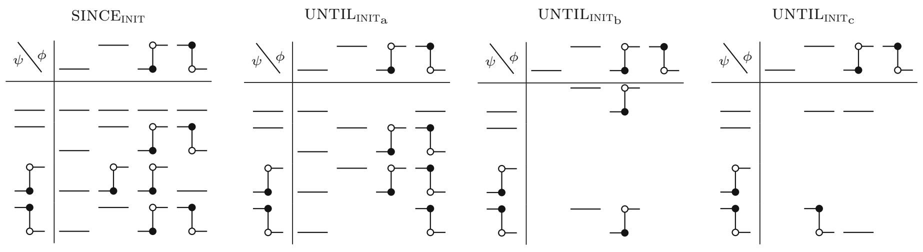

SINCE$_{\mathrm{INIT}}$
UNTIL$_{\mathrm{INIT_a}}$
UNTIL$_{\mathrm{INIT_b}}$
UNTIL$_{\mathrm{INIT_c}}$
ψ
ϕ

Fig. 7. Initialization Tables for the *Since* and *Until* Operators

7 Monitoring Algorithm Complexity

7.1 Instantaneous Transducers

The instantaneous transducers are defined as those for which all emitted transitions take place at the current time, that is, with the same τ as that of the event that caused them. They comprise $\neg$, $\wedge$, $\mathcal{S}$, and $\Diamond_{[-a,0]}$.

The UPDATE procedure can be simplified in that there is no need to keep track of the *valid* intervals. Without the potential for delayed output, the queues will never grow larger than one element and can be replaced with single values.

Theorem 2. *The runtime to monitor expression ϕ that consists of only instantaneous operators on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for $n =$ the sum of the number of transitions on all inputs.*

Proof. The proof is provided in the full version of this paper [1], but is reasonably clear from the pseudocode if it is given that ENQUEUE, DEQUEUE, and SYNC run in constant time.

Theorem 3. *The space required to monitor expression ϕ that consists of only instantaneous operators on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|)$, for $n =$ the sum of the number of transitions on all inputs.*

[4] The proof of the theorems in the following sections are provided in the full version of this paper [1].

Proof. The proof is provided in the full version of this paper [1] and centers on a proof that the queues for the instantaneous transducers do not grow larger than one element.

7.2 Strictly Past Transducer

The transducer for the operator $\Diamond_{[a,b]}$ for $a <= b < 0$ operates identically to that of $\Diamond_{[b-a,0]}$ except that the timestamp of the output that results from an event at time τ is $\tau - b$. The corresponding statement is true for other intervals with open bounds as well. This introduces the need to maintain valid ranges and queues to store the output of the intermediate stages to support operators with multiple inputs.

Theorem 4. *The runtime to monitor expression ϕ that consists of only past-time operators on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for n = the sum of the number of transitions on all inputs.*

Theorem 5. *The space required to monitor expression ϕ that consists of only past-time operators with $\Diamond_{[a,b]}$ where $a < b < 0$ on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi| \left\lfloor \frac{a}{b-a} \right\rfloor)$, for n = the sum of the number of transitions on all inputs.*

Theorem 6. *The space required to monitor expression ϕ that consists of only past-time operators with $\Diamond_{[a,a]}$ where $a < 0$ on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for n = the sum of the number of transitions on all inputs.*

7.3 Restricted Future

The next increment in monitor complexity introduces the metric eventually operator $\Diamond_{[a,b]}$ with $b > 0$. From the UPDATE procedure, we see that it introduces a delay in its output events relative to the input events that produces them. This adds no computational complexity, but reduces the guarantees that can be made about space complexity even when $a \neq b$.

Theorem 7. *The runtime to monitor expression ϕ that consists of past-time and restricted-future operators on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for n = the sum of the number of transitions on all inputs.*

Theorem 8. *The space required to monitor expression ϕ that consists of only past-time operators with $\Diamond_{[a,b]}$ where $a <= b$ and $b > 0$ on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for n = the sum of the number of transitions on all inputs.*

7.4 Unrestricted Future

The introduction of the *until* operator presents two challenges related to the fact that it may remain in an indeterminate state for an arbitrary length of time. The unchanged space complexity belies the fact that whereas a bound may be placed on the growth of the size of the monitor for restricted-future operators if a limit can be placed on the number of transitions within any time interval, no such limit can be placed on the size of the monitor for unrestricted-future operators. Also, it is possible to construct liveness properties that can not be falsified by a monitoring procedure.

Theorem 9. *The runtime to monitor expression ϕ that consists of past-time and restricted-future operators on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for n = the sum of the number of transitions on all inputs.*

Theorem 10. *The space required to monitor expression ϕ that consists of only past-time operators with $\Diamond_{[a,b]}$ where $a <= b$ and $b > 0$ on input signals $\hat{\gamma}$ from time zero until time τ is in $O(|\phi|n)$, for n = the sum of the number of transitions on all inputs.*

8 Conclusion

We have presented straightforward procedures for monitoring dense-time continuous-semantics MTL formulae as well as the tradeoffs in runtime and space complexity incurred as the expressiveness of the supported formulae increases.

We have included the unrestricted-future operators to demonstrate support for full MTL but also because policy writers may find them to be the most natural way of representing the policy that they wish to enforce. That said, they must be used with care as they introduce the ability to describe pure liveness properties for which no truth value will ever be determined, such as $\neg\Diamond\neg\Diamond\phi$.

Future work may include mechanisms for trimming the boolean signals of subexpressions that cannot affect the truth of the full monitored expression as well as the augmentation to the unrestricted-future monitoring algorithm to support the extension of the valid-interval for binary operators in cases for which its output value can be determined based on only the one of its inputs for which the valid-interval extends further in time. For example, a conjunction for which one of its input is unknown beyond τ), but the other is known to be *false* over the entire interval $[\tau$, *current time*$)$.

More immediately, we intend to pursue a VHDL implementation of the subset of MTL for which size restrictions can be guaranteed.

Acknowlegements. Jianwei Niu is supported in part by NSF award CNS-0964710 and the UTSA research award TRAC-2008.

References

1. Baldor, K., Niu, J.: Monitoring metric temporal logic with continuous semantics. Technical Report CS-TR-2012-11, UTSA (2012)
2. Basin, D., Klaedtke, F., Zălinescu, E.: Algorithms for Monitoring Real-Time Properties. In: Khurshid, S., Sen, K. (eds.) RV 2011. LNCS, vol. 7186, pp. 260–275. Springer, Heidelberg (2012)
3. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst. 8(2), 244–263 (1986)
4. Havelund, K., Roşu, G.: Synthesizing Monitors for Safety Properties. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 342–356. Springer, Heidelberg (2002)
5. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Systems 2, 255–299 (1990), doi:10.1007/BF01995674
6. Maler, O., Nickovic, D., Pnueli, A.: From MITL to Timed Automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 274–289. Springer, Heidelberg (2006)
7. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57 (1977)
8. Prabhakar, P., D'Souza, D.: On the Expressiveness of MTL with Past Operators. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 322–336. Springer, Heidelberg (2006)