Discretizing Dense Time

Since there are only finitely many regions, it is possible to find a rational point with small denominator as a representative of each region. If there are \( k \) clocks, we need at least \( k + 1 \) different fractional numbers, so we can take \( \delta = \frac{1}{k+1} \) to be the unit of discretization.

The main question is whether we can follow any dense computation by discrete representatives whose clocks assume values of the form \( n\delta \) for some natural \( n \).

Consider the following computation of a system with three clocks:

\[
\langle t_1 : 1.2, t_2 : 1.4, t_3 : 1.6 \rangle \xrightarrow{\text{tick(0.7)}} \langle t_1 : 1.9, t_2 : 2.1, t_3 : 2.3 \rangle \xrightarrow{\text{reset(t3)}} \langle t_1 : 1.9, t_2 : 2.1, t_3 : 0.0 \rangle
\]

If we record the effect of this computation on the ordering of the fractional parts of the clocks, it can be written as

\[
0 < f_1 < f_2 < f_3 \xrightarrow{\text{tick(0.7)}} 0 < f_2 < f_3 < f_1 \xrightarrow{\text{reset(t3)}} 0 = f_3 < f_2 < f_1
\]

Attempting to emulate this behavior with clocks ranging over \( \frac{n}{4} \), we start with \( \langle t_1 : 1.25, t_2 : 1.50, t_3 : 1.75 \rangle \), and immediately see that there is no time increment \( \Delta = \frac{n}{4} \) which will lead to a state satisfying \( 0 < fr(1.50 + \Delta) < fr(1.75 + \Delta) < fr(1.25 + \Delta) \).
The Problem

Simply increasing the resolution of $\delta$ will not solve the problem.

**Claim 15.** There exists a cts $D_3$ with 3 clocks such that, for every discretization unit $\delta$, there exists a computation of $D_3$ which cannot be emulated by a computation over the time domain $\{n\delta|n \geq 0\}$.

By using combinations of tick steps and resets, the pathologic computation of $D_3$ switches back and forth between states satisfying $0 < f_1 < f_2 < f_3$ and states satisfying $0 < f_1 < f_3 < f_2$, where the difference $f_2 - f_1$ is preserved in the transition from $0 < f_1 < f_2 < f_3$ to $0 < f_1 < f_3 < f_2$ and the difference $f_3 - f_1$ is preserved in the transition from $0 < f_1 < f_3 < f_2$ to $0 < f_1 < f_2 < f_3$. Obviously, these differences get smaller and smaller and, at the end, must become smaller than $\delta$ which is impossible in a $\delta$-discrete computation.
Consider a restricted class of CTS’s in which all inequalities are of the forms $x_i \leq U_i$ or $x_j \geq L_j$. For this class, we can use a significantly more compact discretization.

**Claim 16.** Let $\varphi$ be an assertion using only weak inequalities over the clocks, and let $D$ be a weak-inequalities CTS. Then a state satisfying $\varphi$ is $D$-reachable under the dense time model iff it is $D$-reachable under integer time.

This claim states that integer-time provides a faithful simulation of dense time.

Clearly, if a $\varphi$-state is reachable under integer time, it is also reachable under dense time. It remains to show that any dense-time computation can be simulated by an appropriate integer-time computation.

Let $\sigma : s_0, s_1, \ldots, s_m$ be a dense-time computation leading to the state $s_m$ such that $s_m \models \varphi$. Let $\bar{t}^0 = 0, \bar{t}^1, \ldots, \bar{t}^m$ be the values of the clocks at the states $s_0, \ldots, s_m$, respectively. In particular, let $T^0, \ldots, T^m$ be the values of the master clock at these states. For each $i < m$, let $\gamma_i(\bar{t})$ be the condition on the clocks which allowed the computation to move from $s_i$ to $s_{i+1}$. We also take $\gamma_m = \varphi$. 
Proof Continued

Obviously, for every $i < m$ and every clock $t_j \in C$, the value $t^i_j$ can be expressed as a difference $t^i_j = T_i - T_k$ where $s_k$ is the state on whose entry clock $t_j$ was last reset. In case $t_j$ was not reset before state $i$, $T_k = T_0 = 0$. Thus, we can translate each $\gamma_i(t)$ into a boolean combination of constraints stating weak upper and lower bounds on differences of the form $T_i - T_k$. Satisfying all of these conditions will yield a sequence of values $T_0, \ldots, T_m$ which correspond to entry times of a feasible computation leading to $s_m$.

The fact that $\sigma : s_0, s_1, \ldots, s_m$ is a dense computation shows that the resulting set of inequalities has a rational solution. We will proceed to show that if a set of weak inequalities of the form $L_{ik} \leq T_i - T_k \leq U_{ik}$ has any solution, it also has an integer solution. This will establish the existence of an integer sequence $T_0 \leq T_1 \leq \cdots \leq T_m$ from which we can construct an integer computation leading to a state $\tilde{s}_m$ which satisfies $\gamma_m(t^m) = \varphi(t^m)$. 
A Lemma and Its Proof

**Lemma 1.** Let

\[ S(T_0, \ldots, T_k): \{0 \leq L_{ik} \leq T_k - T_i \leq U_{ik} \mid k \leq i\} \]

be a set of inequalities. Then, \( S \) has a solution iff it has an integer solution.

Assume that we constructed the graph representation \( G \) of the system \( S \). Without loss of generality, assume that \( T_0 = 0 \) and every other variable \( T_i \) for \( 0 < i \leq m \) has the constraint \( 0 \leq T_i \). Note that since all inequalities are weak, all edges will be dashed.

Apply the process of **tightening** to the graph \( G \). We claim that since the system \( S \) is consistent (has a solution) the process of tightening must terminate. Assume to the contrary, that the process does not terminate. Observe that, whenever we replace a label of an edge \( e_{ij} \) by a bigger label \( c_{ij} \), this implies that we have identified a path from node \( T_i \) to node \( T_j \) such that the sum of its (original) labels equals \( c_{ij} \). Note also that this update implies that the constraint \( T_j - T_i \geq c_{ij} \) is a consequence of the original constraints.

Assume that the tightening process fails to terminate, because the label of \( e_{ij} \) has been updated infinitely many times. Let \( \pi_1, \pi_2, \ldots \) be an infinite enumeration of all the paths leading from \( T_i \) to \( T_j \) sorted by increasing length. If we compare the accumulated weights of these paths, we find that infinitely often we identify a path which has a bigger weight than all of its predecessors. In particular, this implies that there exist two paths \( \pi_m \) and \( \pi_n \), with \( m < n \) and \( w(\pi_n) > w(\pi_m) \), such that \( \pi_n \) contains a cycle whose removal would reproduce \( \pi_m \). It follows that the removed cycle has a positive weight, implying that \( S \) can have no solution. Since we know that \( S \) has a (rational) solution, this is impossible.
We therefore conclude that the tightening process terminates. When it terminates, we have that \( c_{ik} \geq c_{ij} + c_{jk} \) holds for every \( i, j, \) and \( k \). Furthermore, if the original constraints were of the form \( T_k - T_i \geq L_{ik} \), then the final values of \( c_{ik} \) satisfies \( L_{ik} \leq c_{ik} \).

We are now ready to identify the solution. We take \( T_0 = 0 \) and, for every \( i > 0 \), \( T_i = c_{0i} \). We will show that these values for \( T_i \) satisfy each of the original constraints. Consider, for example, an original constraint \( T_k - T_j \geq L_{jk} \). Substituting the values for \( T_k \) and \( T_j \), we obtain \( c_{0k} - c_{0j} \geq c_{jk} \geq L_{jk} \).
A General Solution to the Discretization Problem

A solution is provided by allowing system $D_\delta$ to perform also an adjustment step. This is any step which leads from a state $s$ to an equivalent state $s'$, possibly changing the sizes of some of the differences $f_i - f_j$, but preserving their signs. Following is an example of a $0.25$-computation which uses adjustment steps to emulate the run

$$0 < f_1 < f_2 < f_3 \xrightarrow{\text{tick}(0.7)} 0 < f_2 < f_3 < f_1 \xrightarrow{\text{reset}(t_3)}$$

$$0 = f_3 < f_2 < f_1$$

The computation is given by

$$\langle t_1 : 1.25, t_2 : 1.50, t_3 : 1.75 \rangle \xrightarrow{\text{tick}(0.5)} \langle t_1 : 1.75, t_2 : 2.00, t_3 : 2.25 \rangle \xrightarrow{\text{adjust}} \langle t_1 : 1.75, t_2 : 2.25, t_3 : 2.50 \rangle \xrightarrow{\text{reset}(t_3)} \langle t_1 : 1.75, t_2 : 2.25, t_3 : 0.00 \rangle$$

The $\text{tick}(0.5)$ step move the clocks to a temporary situation in which $0 = f_2 < f_3 < f_1$, which does not explicitly appear in the original computation. We then apply an adjustment step which brings us to the situation $0 < f_2 < f_3 < f_1$ as required by the original computation. Now, we can reset $t_3$ to get $0 = f_3 < f_2 < f_1$. 
The General Case

Claim 17. Let \( D \) be a system with \( k \) clocks. Then the system \( D_\delta \) with \( \delta = \frac{1}{k+1} \), allowing adjustments, is region-equivalent to \( D \).

We will show how \( D_\delta \) can emulate every elementary tick move, i.e. a move from \((\bar{d}, \bar{t})\) to \((\bar{d}, \bar{t} + \Delta)\), where \((\bar{d}, \bar{t} + \tau)\) is stable for every \( \tau, 0 < \tau < \Delta \). Let \( f_M \) denote the largest fractional part of a clock in state \((\bar{d}, \bar{t})\). We consider several cases:

- \((\bar{d}, \bar{t} + \Delta)\) is transient: In that case, we let \( D_\delta \) step from \((\bar{d}, \bar{t}^*)\) to \((\bar{d}, \bar{t}^* + \Delta^*)\), where \( \Delta^* = 1 - f_M^* \).

- \((\bar{d}, \bar{t})\) is transient and \((\bar{d}, \bar{t} + \Delta)\) is stable: Let \( 0 = f_0^* < \cdots < f_{r-1}^* \) be a contiguous block of fractional parts such that there exists no \( t_j^* \) with \( fr(t_j^*) = \frac{r}{k+1} < 1 \). Since there are \( k \) clocks, there always exists such a block. We let \( D_\delta \) perform an adjustment step in which every \( t_i^* \) such that \( fr(t_i^*) < \frac{r}{k+1} \) is incremented by \( \frac{1}{k+1} \), while every \( t_i^* \) such that \( fr(t_i^*) > \frac{r}{k+1} \) preserves its value. It is not difficult to see that the resulting \((\bar{d}, \bar{t}^*)\) is equivalent to \((\bar{d}, \bar{t} + \Delta)\).

- Both \((\bar{d}, \bar{t})\) and \((\bar{d}, \bar{t} + \Delta)\) are stable: In that case, \( D_\delta \) does not change its state. Equivalently, it takes a tick step with \( \Delta^* = 0 \).
An Example: The Train Gate Controller – Architectural Layout

We consider a system modeling a railroad crossing. Assume a railroad track crossing a road, where the intersection is guarded by a gate, which is supposed to be down when the train is crossing. In our description, we model the train as one CTS, the physical gate as another CTS, and there is also a controller which senses the approach of the train and instructs the gate to lower or be raised. All reactions and decisions are expected to take some time.

The three components communicate among themselves by shared boolean variables, such as \textit{appr}, \textit{exit}, \textit{lower}, and \textit{raise}. A component \textit{generates} a signal by setting the boolean variable to 1. When the receiver of the signal \textit{senses} that the variable is 1, it responds and reset the variable back to 0.
The **Train Gate Controller** – Behavioral Description

- \( \ell_0 : x > 2 \) \( \xrightarrow{(\text{appr}, x)} (1, 0) \) \( \ell_1 : x < 5 \) \( \xrightarrow{x > 2} \) \( \ell_2 : x < 5 \)

- \( k_0 : \neg \text{appr} \) \( \xrightarrow{\text{appr} / (\text{appr}, z) := (0, 0)} k_1 : z < 1 \)

- \( m_0 : \neg \text{lower} \) \( \xrightarrow{\text{lower} / (\text{lower}, y) := (0, 0)} m_1 : y < 1 \) \( \xrightarrow{y > 1} m_3 : y < 2 \) \( \xrightarrow{\text{raise} / (\text{raise}, y) := (0, 0)} m_2 : \neg \text{raise} \)
Implementation

MODULE main
DEFINE
    d := 4;
    Prog := Tr.prog & Ga.prog & Co.prog;
    train_in := (Tr.loc = 2);
    up := (Ga.loc in {0,1});
    down := (Ga.loc in {2,3});
    Maxb := 5;   Maxc := Maxb*d + 1;
VAR
    x: 0..Maxc;  y: 0..Maxc;  z: 0..Maxc;
    appr: boolean;
    sexit: boolean;
    lower: boolean;
    raise: boolean;
    Tr : process train(x,appr,sexit,d,Maxc);
    Ga : process gate(y,lower,raise,d,Maxc);
    Co : process contr(z,appr,sexit,lower,raise,d,Maxc);
    Idle : process MI;
    Adj : process adjust(x,y,z,Prog,d,Maxc);
    unit : process tick(x,y,z,Prog,d,Maxc);

MODULE MI

Real Time and Hybrid Systems, NYU, Fall, 2003
Module train

MODULE train(x, appr, sexit, d, Maxc)

DEFINE
   prog := loc = 0 | x < 5*d;

VAR
   loc: 0..3;

ASSIGN
   init(x) := 0; init(loc) := 0;
   init(appr) := 0; init(seexit) := 0;
   next(loc) := case loc = 0 & x > 2*d : 1;
   loc = 1 & x > 2*d : 2;
   loc = 2 : 3;
   loc = 3 : 0;
   1 : loc;
   esac;

   next(x) := case loc = 0 & next(loc) = 1 : 0;
   loc = 3 & next(loc) = 0 : 0;
   1 : x;
   esac;

   next(appr) := case loc = 0 & next(loc) = 1 : 1;
   1 : appr;
   esac;

   next(seexit) := case loc = 3 & next(loc) = 0 : 1;
   1 : sexit;
   esac;
Module gate

MODULE gate(y,lower,raise,d,Maxc)

DEFINE
    prog := (loc = 0 & !lower) | (loc = 1 & y < 1*d) |
         (loc = 2 & !raise) | (loc = 3 & y < 2*d);

VAR
    loc: 0..3;

ASSIGN
    init(y) := 0;    init(loc) := 0;
    next(loc) := case  loc = 0 & lower : 1;
                  loc = 1   : 2;
                  loc = 2 & raise : 3;
                  loc = 3 & y > 1*d : 0;
                  1 : loc;
     esac;

    next(y) := case  loc = 0 & next(loc) = 1 : 0;
                  loc = 2 & next(loc) = 3 : 0;
                  1 : y;
     esac;

    next(lower) := case loc = 0 & next(loc) = 1 : 0;
                    1 : lower;
     esac;

    next(raise) := case loc = 2 & next(loc) = 3 : 0;
                   1 : raise;
     esac;
Module contr

MODULE contr(z, appr, sexit, lower, raise, d, Maxc)

DEFINE
  prog := (loc = 0 & !appr) | (loc = 1 & z < 1*d) | (loc = 2 & !sexit) | (loc = 3 & z < 1*d);

VAR loc: 0..3;

ASSIGN
  init(z) := 0;  init(loc) := 0;
  init(lower) := 0;  init(raise) := 0;
  next(loc) := case loc = 0 & appr : 1;
  loc = 1 & z = 1*d : 2;
  loc = 2 & sexit : 3;
  loc = 3 : 0;
  1 : loc;
  esac;
  next(z) := case loc = 0 & next(loc) = 1 : 0;
  loc = 2 & next(loc) = 3 : 0;
  1 : z;
  esac;
  next(lower) := case loc = 1 & next(loc) = 2 : 1;
  1 : lower;
  esac;
  next(raise) := case loc = 3 & next(loc) = 0 : 1;
  1 : raise;
  esac;
  next(appr) := case loc = 0 & next(loc) = 1 : 0;
  1 : appr;
  esac;
  next(sexit) := case loc = 2 & next(loc) = 3 : 0;
  1 : sexit;
  esac;
Module Adjust

MODULE adjust(x,y,z,Prog,d,Maxc)
DEFINE
   d-1 := d - 1;
   free := Prog &
       (x >= Maxc | (x mod d) != nextslot) &
       (y >= Maxc | (y mod d) != nextslot) &
       (z >= Maxc | (z mod d) != nextslot);

VAR
   nextslot : 1..d-1;
ASSIGN
   next(x) := case
       x < Maxc & free & (x mod d) < nextslot : x + 1;
       1 : x;
   esac;
   next(y) := case
       y < Maxc & free & (y mod d) < nextslot : y + 1;
       1 : y;
   esac;
   next(z) := case
       z < Maxc & free & (z mod d) < nextslot : z + 1;
       1 : z;
   esac;
Module Tick

MODULE tick(x,y,z,Prog,d,Maxc)

ASSIGN

next(x) := case
        x < Maxc & Prog : x + 1;
        1 : x;
    esac;

next(y) := case
        y < Maxc & Prog : y + 1;
        1 : y;
    esac;

next(z) := case
        z < Maxc & Prog : z + 1;
        1 : z;
    esac;
Illustrating the Impossibility of Discretization

To illustrate that fixed quantization cannot lead to faithful discretization, consider the following system:

\[
\begin{align*}
  y &:= 0 \\
  x &:= 0 \\
  0 < x < y < 1 \\
  0 < x < 1 < y < 2
\end{align*}
\]

The properties we wish to establish are:

- There exists a computation which reaches \( \ell_1 \) with clocks satisfying \( 0 < x < y < 1 \).

- From every state satisfying \( \text{at}_\ell \ell_1 \land 0 < x < y < 1 \), it is possible to reach \( \ell_2 \).

None of these properties can be expressed in LTL. This is a good opportunity to introduce some elements of the temporal logic CTL.
The Branching Time Temporal Logic CTL

A CTL formula is constructed out of state formulas (assertions) to which we apply the boolean operators $\neg$ and $\lor$ and the CTL operators:

$$\begin{align*}
\text{EX} & \quad \text{AX} & \quad \text{EF} & \quad \text{AF} & \quad \text{EG} & \quad \text{AG} \\
\text{EU} & \quad \text{AU} & \quad \text{EW} & \quad \text{AW}
\end{align*}$$

CTL operators come in pairs of the form $QT$, where $Q \in \{E, A\}$ is a path quantifier, and $T \in \{X, F, G, U, W\}$ is a path operator. Note the correspondence:

$$\begin{align*}
X = & \quad \bigcirc & \quad F = & \quad \Diamond & \quad G = & \quad \Box
\end{align*}$$

It is possible to add past operators among the path operators.

CTL formulas are evaluated over an FDS $D = \langle V, \Theta, \rho, J, C \rangle$. An infinite sequence of states $\sigma : s_1, \ldots$ is called a run segment of $D$ if $s_{i+1}$ is a $\rho$-successor of $s_i$, for each $i = 1, \ldots$. We say that $\sigma$ originates at $s_1$. 
Semantics of CTL

Given an fds $\mathcal{D} = \langle V, \Theta, \rho, \mathcal{J}, \mathcal{C} \rangle$, we define the notion of a CTL formula $p$ holding at a $V$-state $s$, denoted by $(\mathcal{D}, s) \models p$:

- For an assertion $p$,
  
  $(\mathcal{D}, s) \models p \iff s \models p$

  That is, we evaluate $p$ locally on state $s$.

- $(\mathcal{D}, s) \models \neg p \iff (\mathcal{D}, s) \not\models p$

- $(\mathcal{D}, s) \models p \lor q \iff (\mathcal{D}, s) \models p$ or $(\mathcal{D}, s) \models q$

- $(\mathcal{D}, s) \models \text{EX}p \iff (\mathcal{D}, s') \models p$, for some $s'$ a $\rho$-successor of $s$

- $(\mathcal{D}, s) \models \text{AX}p \iff (\mathcal{D}, s') \models p$, for every $s'$ a $\rho$-successor of $s$

- $(\mathcal{D}, s) \models \text{EF}p \iff$ there exists an $s$-originating run segment $\sigma$

  containing a state $s' \in \sigma$, such that $(\mathcal{D}, s') \models p$

- $(\mathcal{D}, s) \models \text{AF}p \iff$ every $s$-originating run segment $\sigma$

  contains a state $s' \in \sigma$, such that $(\mathcal{D}, s') \models p$

- $(\mathcal{D}, s) \models \text{EG}p \iff$ there exists an $s$-originating run segment $\sigma$

  all of whose states $s' \in \sigma$ satisfy $(\mathcal{D}, s') \models p$

- $(\mathcal{D}, s) \models \text{AG}p \iff$ all states $s' \in \sigma$ on all $s$-originating run segments $\sigma$

  satisfy $(\mathcal{D}, s') \models p$

Similar definitions can be given for the path operators $\text{U}$ and $\text{W}$.
Specifying Existential Properties

Returning to the system of interest:

We can specify

- It is possible to reach a state satisfying $at_{\ell_1} \land 0 < x < y < 1$.

$$\text{EF}(at_{\ell_1} \land 0 < x < y < 1)$$

- From every state satisfying $at_{\ell_1} \land 0 < x < y < 1$, it is possible to reach $\ell_2$.

$$\text{AG}((at_{\ell_1} \land 0 < x < y < 1) \rightarrow \text{EF}at_{\ell_2})$$
Encoding it in TLV

In file `discr.smv`, we put:

```plaintext
MODULE main -- Program discr.smv. Simple counter-example
DEFINE d := 4; Prog := 1;
    Maxb := 2; Maxc := Maxb*d + 1;
VAR x: 0..Maxc; y: 0..Maxc;
    Aut : process automaton(x,y,d,Maxc);
    Idle : process MI;
    Adj : process adjust(x,y,Prog,d,Maxc);
    unit : process tick(x,y,Prog,d,Maxc);
MODULE automaton(x,y,d,Maxc)
VAR loc : 0..2;
    reset: 1..3;
ASSIGN
    init(loc) := 0;
    init(x) := 0; init(y) := 0;
    next(loc) := case loc = 0 & 0<x & x<y & y<d : {0,1};
        loc = 1 & x<d & d<y & y<2*d : 2;
        1 : loc;
    esac;
    next(x) := case loc = 0 & next(loc) = 0 & reset=1 : 0;
        1 : x;
    esac;
    next(y) := case loc = 0 & next(loc) = 0 & reset=2 : 0;
        1 : y;
    esac;
```

Real Time and Hybrid Systems, NYU, Fall, 2003
The Proof Script

In file `discr.pf` we place:

Print "\n Check Viability\n";

Let at1 := (loc=1) & 0<x & x<y & y<d;
Let at2 := (loc=2);

Print "\n It is possible to get to ell_1\n";

Call mctl(cEF(at1));

Print "\n After entering ell_1,";
   " it is always possible to get to ell_2\n";

Call mctl(cAG(at1 -> cEF(at2)));

To check the bad case, we form a file similar to `discr.smv`, except that we comment out the declaration

```plaintext
-- Adj  : process adjust(x,y,Prog,d,Maxc);
```