# Model Checking a real-time Foveate Controller using Timed Automata
Technical Report TR-05-56

Gary Holness
Computer Science Department
140 Governors Drive
University of Massachusetts
Amherst, MA 01003 USA
gholness@cs.umass.edu

**Abstract**

Research activity in the area of smart spaces strives to endow an environment with a myriad of sensory-motor and computational devices which, together with, various learning algorithms may discern useful information about activity within the environment. Computer vision has proved an important sensory mode for uncovering features and dynamics associated with the events that occur. Tracking is an important task in visual sensing. The literature is full of many examples of real-time trackers.

The proliferation of embedded computation grounded in physical systems will continue for the foreseeable future. Helping to increase the correctness of such systems means provable validation of real-time constraints. In this paper we present a system that employs the Timed Automata formalism to verify a foveate controller designed as a processing pipeline.

## 1 Introduction

In the human visual system, there exists a small pit-like area in the center of the retina called the *fovea centralis*. In a healthy adult, image information from this region is most sharp and detailed. Biologically, this makes sense as objects right in front of you are within arm's reach and available for manipulation. More information about an object allows a person to better manipulate it.

In computer vision, image acquisition begins with an array (image plane) of picture elements (pixels) which produce measurable intensity levels in response to various frequencies of light. Inspired by human vision, more complex image processing can be devoted only to the central region of the image plane. In doing so, we see a computational savings and are able to achieve higher effective processing rates. When tracking a subject, the interesting object is the moving target. Thus, it is the goal of a tracking system to maintain the moving subject in the foveal region of the image plane.

## 2 Control

Getting physical systems to behave in stable ways is the domain of a control system. In its most general form (figure 1), a feedback control system consists of four components, namely a system under control, and input reference, a measurement, and a decision process. The system under control is our plant. This system has a set of controllable variables which can be manipulated by injecting energy into the system. The input reference serves as a desired goal for the plant to achieve. The measurement device (or sensor) supplies feedback about the current state of the plant. The feedback is compared to the input reference producing an error signal used as the basis for the decision process or control law. The result of this computation is an output signal used to inject energy (or actuate) the plant so as to manipulate its controllable variables.

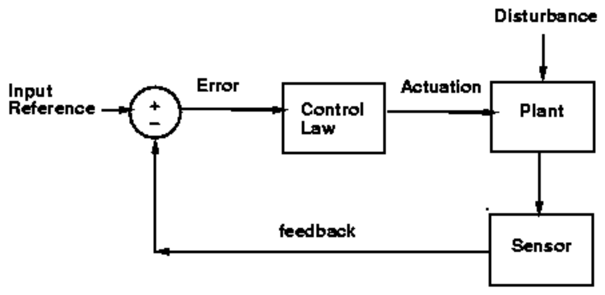


Figure 1: General schematic of a feedback control system

Also affecting the state of the plant are external disturbances that add or remove energy from the plant. It is the goal of the feedback (or closed loop) controller to act in a fashion so as to reduce error to zero. By continually acting in a sense/act/respond loop, a closed loop controller is able to reject disturbances. An example closed loop controller is a thermostat heating system. In this example the room is the plant, the controllable variable is ambient temperature, we have a heat sensor providing feedback, and the thermostat set point is the input reference. The controller actuates the plant using a furnace. Disturbances in this system are thermal dissipation through walls, windows and doors as well as thermal energy radiated by the bodies of the room's inhabitants. Depending on the application, the way in which a controller advances system state toward the goal is important. For example, if some notion of comfort is a concern, a successful controller will give rise to smooth trajectories. Controllers are categorized based on what the decision process computes as a function of error [6]. For example, in proportional (or P) control, the control law computes an actuation signal $f_{a_{t+1}}(e_t) = ke_t$ at time $t$ which is a constant function of the error signal at time $t$. This actuation signal causes a change in plant state, $\Delta s_t$. So, we have $s_{t+1} = s_t + \Delta s_t$. The next state is some function $s_{t+1} = g(s_t, f_{a_{t+1}})$ of the current state and actuation signal. The change in state of the plant is governed by its dynamics which arise from physical properties. Thus, the function $g$ describes the response of the plant when in a configuration $s_t$ and injected with energy $f_{a_{t+1}}$. Other types of control include proportional derivative (PD) and proportional integral derivative (PID) control. From a dynamical systems perspective [15, 12], the system response for the controller is a *map.* Plotting the state of the plant for time steps in infinity results in a *phase portrait* through state space that captures the behavior of our plant over time. The plant starts in an initial configuration $s_0$ with a goal reference of $s_0$. A controller is characterized by its map's trajectories through phase space as $\Delta s_t$ settles to zero for $t = 0, 1, \ldots$. Given a $n$-dimensional phase space $S = (s_1, \ldots, s_n)$, the control law becomes a map $M : S \mapsto S$. For positive integer, $k$, define map $M^{(k)}$ by the $k$-fold iteration of $M$ with itself. A class of fixed points (figure 2) $S_a$ or *attractors* are such that

$$\lim_{k \to \infty} \left( M^{(k)}(S_i) \right) = S_a$$

Where $S_i$ is the set of initial configurations (neighborhood or basin of attraction) from which system state converges to $S_a$. A *repeller* $S_r$ is a fixed point from which system state diverges as we iterate a map.

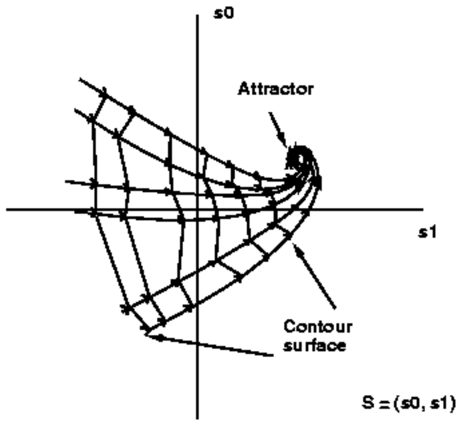


Figure 2: Contour surface in phase space

Points along trajectories through phase space describe contour lines or a potential surface (figure 3) defining the control objective. Control decisions are made by gradient descent along the potential surface. A control system that generates stable behavior is one for which system state converges to an

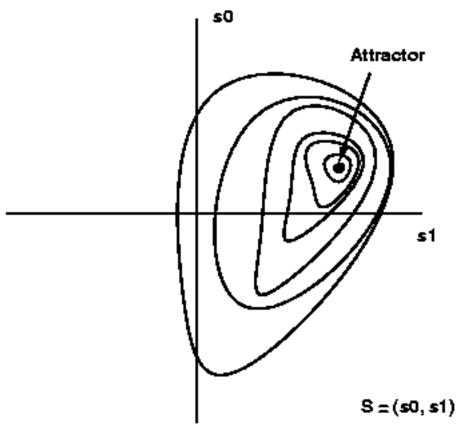


Figure 3: Potential surface for controller

attractor from initial conditions of interest. An example of this is the range of safe operating temperatures for a climate control system.

Techniques such as Lyopanov's method [12] formulate plant state in terms of energy functions and their time based derivative. In this approach, if energy dissipates over time to zero, then the system is stable. System response evolves at a characteristic rate. For example, heat dissipates from a room depending on the building materials and insulation. Thus, processes that measure and respond in a control system must run at a characteristic rate in a real-time system. Reasoning about stability does not imply the controller software actually achieves its real-time guarantees as it moves system state toward the attractor. With formal verification, a guarantee on real-time constraints while moving system state from one level curve to the next is achieved. Moreover, by structuring the software appropriately, such systems are more easily realizable.

## 3 Foveate Controller

Our system consists of a pan/tilt/zoom (PTZ) camera interfaced to a computer host equipped with a frame grabber card. Acquired images are obtained from the image plane from a special area in memory. The camera is trained on a scene in an in-door laboratory environment. We have a moving object in a scene that we find interesting and wish to track. Raw data from the image plane is processed to extract pixels belonging to the object of interest. This processing involves classifying and extracting foreground pixels from background pixels, segmenting pixels into objects and computing the centroid of the objects. Simple background subtraction is used to compute the foreground pixels.

From the camera we get a sequence of frames (image plane data). For each pixel $p_{ij}$, we maintain a window of $n$ observations $< p_{ij_1}, \ldots, p_{ij_n} >$. Define a function $Sig$ which measures variability across a window of data. For each pixel, if $Sig(< p_{ij_1}, \ldots, p_{ij_n} >) \leq \epsilon$ for some acceptable level $\epsilon$ , then any change in $p_{ij}$ from frame 1 to frame $n$ is considered noise. This being the case, we can build an up to date model of the background $B$ where $B_{ij} = \{p_{ij_n} | Sig(< p_{ij_1}, \ldots, p_{ij_n} >) \leq \epsilon\}$. The function $Sig$ can be computed many ways. A very simple approach measures, $diff(B_{ij}, p_{ij})$, how much a pixel changed between the current background and the most recent frame. If this is below a threshold, then the new pixel is considered a noisy background pixel. Another approach measures whether or not the most recent pixel's value falls outside, $\bar{X} + k\hat{\sigma}$, some multiple of the sample standard deviation outside of the sample mean. Given a current background model $B$, we compute the foreground $F$ by $F_{ij} = p_{ij} - B_{ij}$. Given an array of foreground pixels, the next problem is to classify pixels as belonging to a particular object. One approach is using connected components analysis [8]. By this approach, each pixel $p_{ij}$ is considered a node. A pixel $p_{ij}$ shares an edge with a pixel $p_{kl}$ if $p_{ij}$ and $p_{kl}$ satisfy some spatial relationship. Two popular relationships are the $FourNeighbor$ and $EightNeighbor$ operators where...

$$\begin{aligned} FourNeighbhor(p_{ij}, p_{kl}) &= North(p_{ij}, p_{kl}) \vee South(p_{ij}, p_{kl}) \\ &\quad \vee East(p_{ij}, p_{kl}) \vee West(p_{ij}, p_{kl}) \\ EightNeighbor(p_{ij}, p_{kl}) &= FourNeighbor(p_{ij}, p_{kl}) \vee \\ &\quad NorthEast(p_{ij}, p_{kl}) \vee \\ &\quad SouthEast(p_{ij}, p_{kl}) \vee \\ &\quad NorthWest(p_{ij}, p_{kl}) \vee \\ &\quad SouthWest(p_{ij}, p_{kl}) \end{aligned}$$

By connected components analysis, we iterate over the pixels in the image and label them as belonging to the same object if they are in the same connected component. Once the object has been segmented, a bounding box is circumscribed with corner points $(x_{min}, y_{min})$, $(x_{max}, y_{min})$, $(x_{max}, y_{max})$, and $(x_{min}, y_{max})$ where $x_{min} = min_{x-coord}\{p_{ij} \in F\}$ likewise for $x_{max}$, $y_{min}$, and $y_{max}$. The centroid $(c_x, c_y)$ is computed as $c_x = x_{min} + \frac{1}{2}(x_{max} - x_{min})$ and $c_y = y_{min} + \frac{1}{2}(y_{max} - y_{min})$.

The goal of the foveate controller is to maintain the foreground object of interest in the center of the image plane. In our control system, the plant is the room containing the target and the camera. The sensor is the image plane from which we extract the centroid of the moving object. The plant is actuated by sending commands to the motors in the camera causing it to pan or tilt. The input reference is the coordinate of the center of the image plane $(I_{c_x}, I_{c_y})$. The vector valued error signal $(e_x, e_y)$ is computed (figure 4) by taking the difference $e_x = I_{c_x} - c_x$ and $e_y = I_{c_y} - c_y$.

The cameras are actuated using position control. That is, given a position error the control law computes an actuation signal for velocity to pan/tilt the camera as quickly as possible. The control law implemented uses proportional derivative (PD)control where the control law computes an actuation sig-

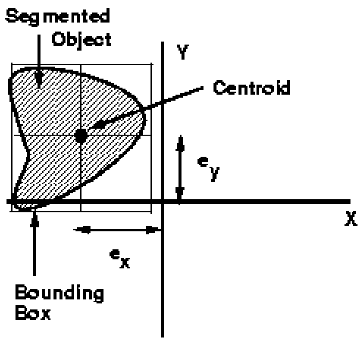


Figure 4: Tracking error in foveate controller

nal $f_{a_{t+1}}(e_t) = k_d \frac{\partial e_t}{\partial t} + k_p e_t$. The constants $k_p$ and $k_d$ are the proportional and derivative gains respectively.

## 4 Programming Paradigm: Event Listener Pipeline

We wanted to formulate the design in a way that lends itself well to formal verification. Borrowing from how the Java programming language [9] implements structured event handling, we implemented control system components in C++ using the EventListener pattern. In the EventListener pattern, there are two roles, namely the EventListener (or listener) and the EventGenerator (or generator). In object oriented programming, a class specifies the layout of an object. An object has instance data which comprise the object's state and methods which operate on that data. Methods fall into two categories, namely *accessors* which examine data and *mutators* which modify that data. As a computation proceeds and mutators on an object are called, the object's state sequences through a number of state changes $d_0, d_1, d_2, \ldots$ (figure 5). Let $d_t$ represent an object's state at time-step $t$. One of the tenets of object oriented programming is that all interaction with an object's state is done through the accessors and mutators.

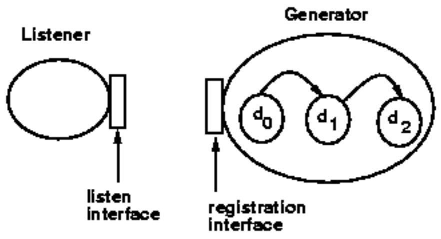


Figure 5: State changes in event generator

Thus, an object's instance data cannot be directly observed. There are cases where a process should take some action in response to an object's state changes. Without ability to monitor the object's state directly, another mechanism is needed. In the EventListener pattern, a generator makes available a registration interface which allows listeners to express interest in particular state changes $< d_i, d_j >$. A listener makes available an interface which allows generators to contact it with an event. Interest is expressed by explicit registration by the listener.

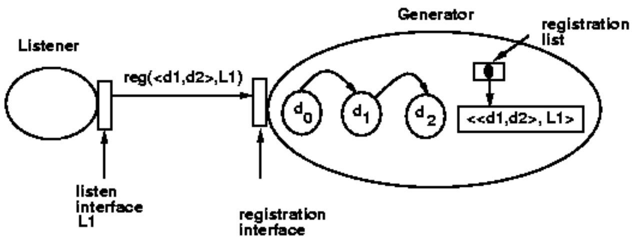


Figure 6: Listener registers with the generator

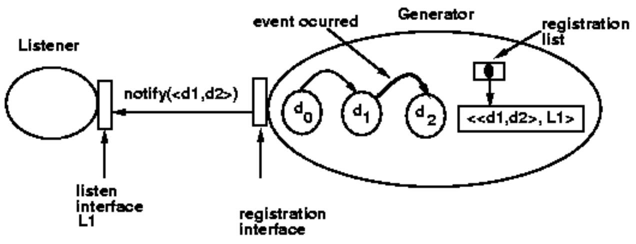


Figure 7: Generator notifies the listener

When a listener registers (figure 6), it specifies the state change or Event $< d_i, d_j >$ of interest and a handle $L_k$ by which it can be contacted. The generator maintains a map $<< d_i, d_j >, L_k >$ of registrations. Whenever event $< d_i, d_j >$ occurs in the generator, the appropriate set of listeners are notified (figure 7). The interfaces for EventListeners and Generators and Events are as follows...

```
class EventListener {
  public:
    EventListener();
    virtual ~EventListener();
    virtual bool notify(Event *event)
                     {return false;};
    virtual bool equals(EventListener * listener)
                       { return false;};
};

class EventGenerator {
  public:
   EventGenerator();
   virtual ~EventGenerator();
   virtual bool registerListener(EVENT_TYPE type,
                                 EventListener *listener)
                        {return false;};

   virtual bool equals(EventGenerator * generator)
                        {return false;};
};

class Event {
  public:
    Event();
    virtual ~Event();
    virtual EVENT_TYPE getType()
                 {return EVENT_NONE;};

    virtual bool equals(Event *data){ return false;};
};
```

The entire control system was implemented by chaining together controller components as a sequence of event generators and listeners in a pipeline. Such a design allows for benchmarking of each component for its timing requirements. Some components assume only the single role of a generator, while others assume the dual role of both listener and generator. As a generator, an object allows listeners to register for particular events. As a listener an object is able to be notified of particular events. Objects implementing the event interface can include additional data or methods. In our system, this data includes the result of a processing stage. Table 1 lists each component, their overall function, and their listener and or generator role.

## 5 Timed Automata

Timed automata, introduced by Alur and Dill [2], is the most widely studied theoretical model for verification of real-time systems. A timed automaton is a finite automaton augmented with real-valued clocks. The semantics are such that transitions are instantaneous and time may elapse in a state (or location). During a transition, some of the clocks may be reset. The value of a clock may be read at any instant. A clock's value represents the time which has

| **Component** | **Generator Role** | **Listener Role** |
|---|---|---|
| MeteorFrameGrabber | EVENT_RGB_DATA | NONE |
| CroppingImagePlane | EVENT_RGB__DATA | EVENT_RGB_DATA<br>EVENT_VIRTUAL_SERVO |
| BackgroundSubtractor | EVENT_RGB_DIFF_DATA | EVENT_RGB_DATA |
| DynamicBackgroundSubtractor | EVENT_RGB_DIFF_DATA | EVENT_RGB_DATA |
| ObjectSegmenter | EVENT_BLOB_DATA | EVENT_RGB_DIFF_DATA |
| CentroidComputer | EVENT_CENTROID_DATA | EVENT_BLOB_DATA |
| ErrorComputer | EVENT_POSITION_ERROR_DATA | EVENT_CENTROID_DATA |
| ControlLaw | EVENT_SERVO | EVENT_POSITION_ERROR_DATA |
| EviCamera | NONE | EVENT_SERVO |
| VirtualCamera | EVENT_SERVO | EVENT_POSITION_ERROR_DATA |

Table 1: Software Components and their Generator/Listener Roles

lapsed since the last time the clock was reset. There are constraints on clocks expressed in the form of an inequality. These constraints occur in two forms, namely constraints on transitions and constraints on locations in the automaton. he former are called guards and the latter are called invariants. A guard has semantics such that the associated transition can only be taken if the current clock values satisfy the guard. An invariant has semantics such that time can elapse in the location as long as the current clock values satisfy the invariant.

## 5.1 Syntax

Let $X$ be a set of clock variables ranging over $\mathbb{R}^+$. We have constraints $\mathcal{C}(X)$ are inequalities of the form $x \prec c \in \mathcal{C}$ where...

- $\prec = \begin{cases} < \\ \leq \end{cases}$
- $c$ is a non-negative rational
- If $\phi_1, \phi_2 \in \mathcal{C}$
  then $\phi_1 \wedge \phi_2 \in \mathcal{C}$

Define a timed automaton $A = (\Sigma, S, S_0, X, I, T)$ where...

- $\Sigma$ is finite alphabet
- $S$ is finite set of locations
- $S_0$ is finite set of starting locations
- $X$ is set of clocks
- $I : S \rightarrow \mathcal{C}(X)$
  is a labeling function assigning invariants to locations
- $T \subseteq S \times \Sigma \times \mathcal{C}(X) \times 2^{|X|} \times S$
  are transitions $< s, a, \phi, \lambda, s' >$ from state $s$ to $s'$,
  labeled $a$, with guard $\phi$ and reset clock set $\lambda$.

## 5.2 Semantics

The model for $A$ is an infinite transition graph $\mathcal{T}(A)$. Define $\mathcal{T}(A) = (\Sigma, Q, Q_0, R)$ where...

- $Q$ the set of states $(s, \nu)$
  where $s \in S$ is a location and $\nu : X \to \mathbb{R}^+$ is a clock interpretation.
- $Q_0 = \{(s, \nu) | s \in S_0 \wedge \forall_x \in X(\nu(x) = 0)\}$
  is the set if initial states.
- R is the transition relation

Given $\lambda \subseteq X$, define $\nu[\lambda = 0]$ as

- clock assignment which agrees with $\nu$ for $X - \lambda$
- maps clocks in $\lambda$ to 0.

and for $d \in \mathbb{R}$, we have

- For $d \in \mathbb{R}$, $\nu + d$ is a clock assignment mapping $x \in X$ to $\nu(x) + d$
- For $d \in \mathbb{R}$, $\nu - d$ is a clock assignment mapping $x \in X$ to $\nu(x) - d$

We have transitions of two types, namely *delay* and *action* transitions. Delay transitions correspond to the elapse of time in some location. Define a delay transition...

- $(s, \nu) \to^d (s, \nu + d)$ where $d \in \mathbb{R}^+$
- $\forall_e$ s.t. $0 \leq e \leq d$, $I(s)$ holds true for $\nu + e$.

This means an invariant holds throughout the delay. Action transitions correspond to taking a transition in $T$. Define an action transition...

- $(s, v) \to^a (s', \nu')$ for $a \in \Sigma$.
- Given there is a transition $< s, a, \phi, \lambda, s' >$ where $\nu \models \phi$ and
- $\nu' = \nu[\lambda = 0]$ or $\nu'$ agrees with $\nu$ on $X - \lambda$

A transition relation $R$ of $\mathcal{T}(A)$ is built by combining delay and action transitions. We have that $(s, \nu)R(s', \nu')$ or $(s, \nu) \to^d (s', \nu')$ if $\exists_{s",v"} s.t(s, \nu) \to^d (s", \nu") \to^a (s', \nu')$ for some $d \in \mathbb{R}^+$. The transition relation $R$ is an association of two states in $\mathcal{T}(A)$ separated by a delay transition and an action transition.

## 5.3 Model Checking

Given a set of constraints on clock variables in $X$ we get a multidimensional shape representing a set of clock assignments or *clock zone*. Verification of a system means testing if the infinite transition graph $\mathcal{T}(A)$ gets to a situation where there is no possible clock interpretation which satisfies it. In other words, the clock zone becomes empty as we take transitions in $\mathcal{T}(A)$. This reduces to doing reachability analysis on $\mathcal{T}(A)$ [1]. By this technique, a check is done by starting in an initial state of $\mathcal{T}(A)$ and following transitions. This results in a path or *execution trace* through the infinite transition graph. If a step along an execution trace results in an empty clock zone, it means there is a flaw in the model. If a previously encountered state $(s, \nu)$ is uncovered while following an execution trace, it means that execution has cycled back to a valid state. When this happens, it means the model is satisfiable and, thus validated.

Operations on clock zones include intersection, reset (or projection) and time lapse [10]. Given that we have two clock zones $\phi$,$\psi$ we have that their intersection $\phi \wedge \psi$ is also a clock zone. Given that we have a clock zone $\phi$ and clocks $\lambda \subseteq X$, we have that their projection $\phi[\lambda = 0]$ is also a clock zone. Given that we have a clock zone $\phi$, we have that, for clock interpretation $\nu$, $\nu \in \phi^{\Uparrow}$ if $\nu$ satisfies $\nu \models \exists_{t \geq 0} [(\nu - t) \in \phi] \vee \exists_{t \geq 0} [(\nu + t) \in \phi]$.

States are represented by zones $(s, \phi)$ where $s$ is a location of $A$ and $\phi$ is a clock zone. If we consider the transition $e = (s, a, \psi, \lambda, s')$, the current zone is $(s, \phi)$. Define the zone $suc(\phi, e)$ as the set of assignments $\nu'$ s.t. for some $\nu \in \phi$, $(s', \nu')$ can be reached from $(s, \nu)$ by letting time lapse and executing a transition $e$. The pair $(s', succ(\phi, e))$ is the set of successors of $(s, e)$ obtained by...

- Intersect current zone $\phi$ with invariant $I(s)$ of location $s$ to find the set of possible clock interpretations for the current state.

- Let time lapse in location $s$ via $\phi^{\Uparrow}$

- Intersecting with $I(s)$ again to find clock interpretations still satisfying $I(s)$.

- Intersect with guard $\psi$ of transition to find clock interpretations permitted by the transition.

- Set clocks in $\lambda$ to 0 to project the clock interpretations onto clocks in $\lambda$.

To perform verification, on the fly, we compute transitions $(s, \phi) \rightarrow^{a} (s', succ(\phi, e))$ continually until we either encounter a repeated state or the clock zone is empty. This gives us an algorithm for determining reachability.

## 5.4 Difference Bound Matrix

A representation for clock zones is the difference bound matrix (DBM) [5]. A DBM is a square matrix indexed by clock variables in $X$. Included is a special clock $x_0$ whose value is always 0. We have $D_{ij} = (d_{ij}, \prec_{ij})$ represents the inequality $x_i - x_j \prec_{ij} d_{ij}$. There are many DBMs for the same clock zone. We produce a normal form of a DBM by tightening it using the $O(n^3)$ Floyd-Warshall Algorithm. By this approach, we have $D_{ik}$ bound by $d_{ik} \prec_{ik} d_{ij} + d_{jk}$. If this does not hold for some $j$, then we replace $D_{ik}$ by $(d'_{ik}, \prec'_{ik})$ where...

$$d'_{ik} = d_{ij} + d_{jk}$$

$$\prec'_{ik} = \begin{cases} \leq & if \prec_{ij} is \leq and \prec_{jk} is \leq \\ < & otherwise \end{cases}$$

Once converted to normal (or canonical) form, we can determine if the clock zone represented by the DBM is empty by examining the main diagonal. If the clock zone is empty or unsatisfiable, then at least 1 entry along the main diagonal $\neq (0, \leq)$.

## 6 System Implementation

Our system consists of a tool written in Java called TACreator (figure 8). Using this system, we draw or *capture* event listener pipeline designs expressed as a timed automaton. By this approach, a node represents a component that implements either a generator or listener role. An edge represents the throwing of an event. Invariants and guards are used to describe the amount of processing time spent in a component and sending events. We label each location and edge with a string. Labels on locations and edges describe component names in the pipeline and events respectively.

The tool is comprised of a drawing palette, and toolbar. In the toolbar, appear three buttons for drawing a final state, non-final state or transition. Selecting one of these sets the context for what will happen in the drawing area. If a non-terminal state is selected, TACreator assumes the first non-terminal state to be drawn is the start state. To draw an edge, the mouse cursor must be clicked in the outbound state and dragged inside the state at the inbound side of the edge. A state is drawn by clicking the mouse in the drawing area. The fundamental unit of rendering in the drawing area is a TASelectable. Start states, final-states, non-final states, and edges all implement a TASelectable interface. A TASelectable interface supports basic methods which allow an object to participate as a renderable object in the drawing region. This includes features such as hi-lighting, mousing over and selecting.

When the mouse cursor is moved over screen real-estate occupied by the shape, it is drawn more bold to connote that it can be selected. When the mouse is clicked on a shape, it is identified as being selected so that information associated with it can be easily obtained. When a shape is selected, visual feedback is given by hi-lighting it yellow. Associated with each TASelectable is a TAPropertySheet. A TAPropertySheet supports all of the information contained in a TASelectable. This includes a TASelectable's associated name, clock constraint etc. In addition to holding data attached to a TASelectable, a TAPropertySheet also supplies the user interface necessary for interacting with

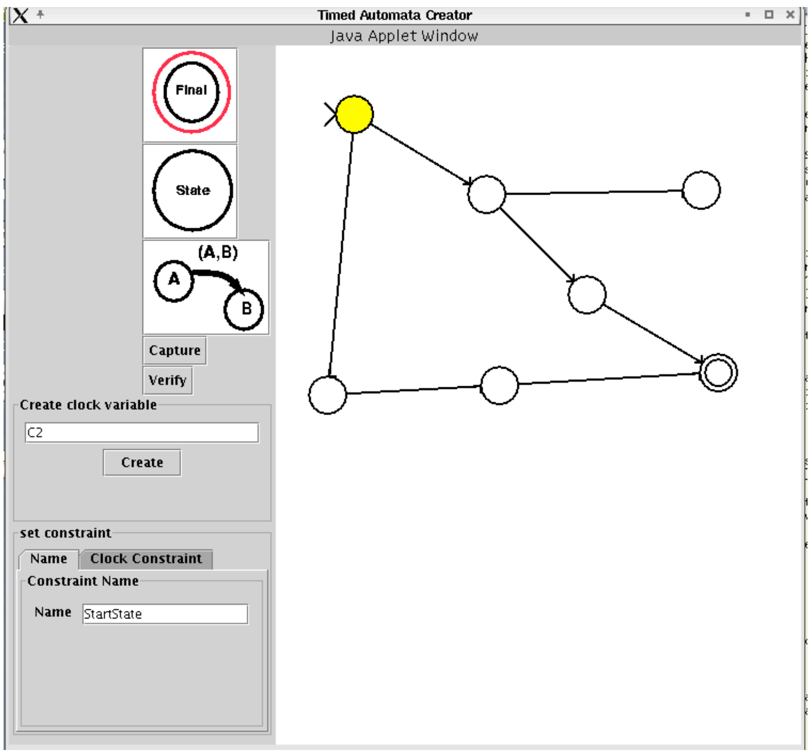
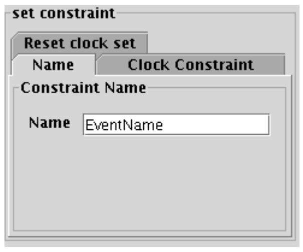


Figure 8: A timed automata specified using TACreator

this data. In the tool bar at the left side of TACreator, at the bottom, the property sheet for the currently selected TASelectable is displayed. With this, a context dependent interface is achieved. As the timed automaton is drawn, new TASelectables are added to a list. Whenever a mouse event occurs, the list is consulted to see who owns the screen real-estate where the mouse event occurred. The owner is then updated appropriately. Key to a timed automaton are clock variables which take on values from non-negative rationals. The tool bar contains an interface for entering new clock variables by typing the clock variable name, hitting enter, and clicking the create button.

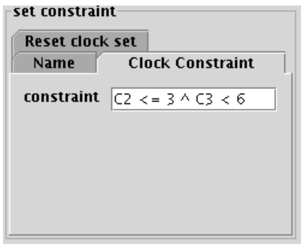
For a selected edge in the timed automaton, its name, guard (clock constraint) and reset clock set are entered using the edge's TAPropertySheet UI (figure 9). Likewise, for a selected location in the timed automaton, its name and invariant (clock constraint) are entered using the location's TAPropertySheet UI (figure 10). In these examples, the edge "EventName" has guard $c_2 \leq 3 \wedge c_3 < 6$ and reset clock set $\lambda = \{c_2, c_3\}$. Also, location "State" has invariant $c_1 \leq 1 \wedge c_2 < 3$. In both cases, clock constraints and names are represented as strings. A reset clock set is represented as a pick list. The list on the left of figure 9c is initialized with the set of available clocks.

This set is produced from the union of all added clocks and the special zero



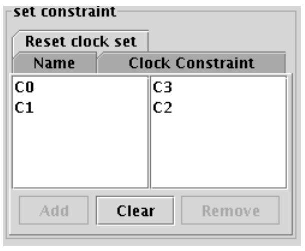
Name EventName

(a) Name



constraint C2 <= 3 ^ C3 < 6

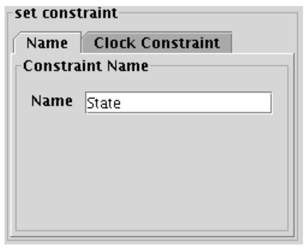
(b) Guard



| | |
|---|---|
| C0 | C3 |
| C1 | C2 |

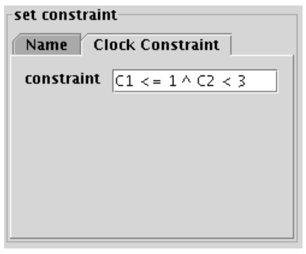
(c) Reset clock set

Figure 9: Name, guard and reset clock set for an edge



Name State

(a) Name



constraint C1 <= 1 ^ C2 < 3

(b) Invariant

Figure 10: Name and invariant for a location

clock. With the exception of the zero clock, all clocks in this list can be moved between the list on the left and the list on the right. The set of clocks in the list on the right is the reset clock set.

The strings representing clock constraints are parsed into a DBM representation. An LALR1 grammar [11] was defined for clock constraints…

```
Formula --> CompoundFormula

CompoundFormula --> SingleFormula |
            SingleFormula Conjunction CompoundFormula

SingleFormula --> SimpleFormula |
                  ComplexFormula

SimpleFormula --> Variable Inequality Number

ComplexFormula -->
   Variable Minus Variable Inequality Number
```

where `Variable`, `Inequality`, `Number`, `Minus`, and `Conjunction` are terminals and `Formula`, `CompoundFormula`, `SingleFormula`, `SimpleFormula`, and `ComplexFormula` are non-terminals. Borrowing from a technique used in the compiler construction tools Lex and Yacc [13], an expanded rule results in a stack of tokens. The two rules of importance are SimpleFormula and ComplexFormula. A SimpleFormula results in the following token stack...

```
0: Variable
1: Inequality
2: Number
```

and a ComplexFormula results in the following token stack...

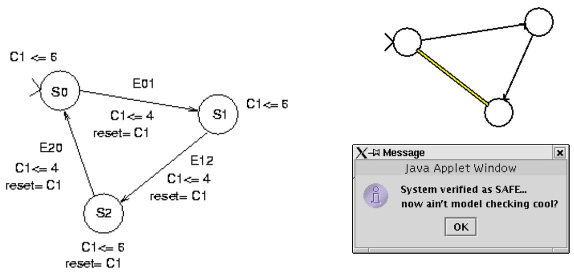
```
0: Variable
1: Minus
2: Variable
3: Inequality
4: Number
```

These token stacks are passed to a semantics routine which build the appropriate difference bound matrix (DBM) representation for the clock constraint. All of the DBM operations, canonicalize, intersect, elapse time, and reset clocks were implemented such that the parameters are not modified and the result is returned by allocating space for a new DBM. This was done rather than producing results in-place because a DBM may need to be re-used when performing reachability search. An additional operation called `isValid` was added. This method tests whether or not a DBM represents an empty clock zone.

Once a controller's timed automata is constructed, this design is *captured* by clicking the capture button in the tool palette. Capturing a design means it is unavailable for modification as it is necessary to have a static design for verification. Verification is performed by clicking the verify button in the tool palette. A design is verified by computing $(s', succ(\phi, e))$ from $(s, \phi)$ continually from the start node. This reduces to search. Three versions of depth first search were implemented with varying assumptions. The first assumes the timed automaton has final states and performs DFS up to the terminal states. The second assumes the control loop has a single cycle and performs DFS until it reaches an empty clock zone, a repeated node $(s', \phi)$ in the model or reaches a maximum depth. The third assumes the control loop has multiple cycles and performs DFS until it reaches an empty clock zone, a repeated node or a maximum depth. Bounding search depth to some maximum was done to avoid running into system limitations. In the case where maximum search depth is reached, the system prompts the user appropriately. In the case where an empty clock zone is reached or the design is successfully verified, the user is also prompted appropriately.

## 7 Results

The system was tested for its ability to find empty clock zones. An example test case is the timed automaton of figure 11(a). The data output for this test case appears in the full technical report [7]. As can be seen in figure 11(b), the system is correctly verified as safe. In this example, the model consists of three states $S_0, S_1, S_2$ and edges $E_{01}, E_{12}, E_{20}$ labeled with invariants and guards involving a single clock variable. The invariants allow time to elapse for up to 6ms and the guards allow transitions to be taken within a deadline of 4ms. This example was chosen for its very tight timing constraints.



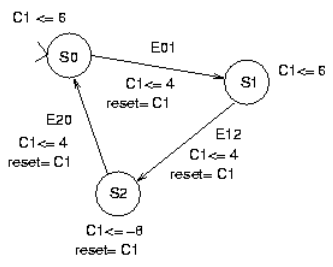
System verified as SAFE...
now ain't model checking cool?

(a) A working test TA

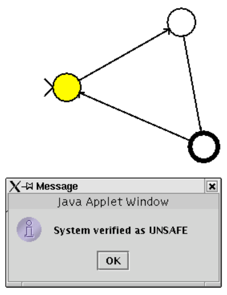
(b) Correct TA verified

Figure 11: Example of a correct design

An example test case is the timed automaton of Figure 12(a). In this example, $S_2$'s invariant was made incompatible with $C_1$'s value after reset due to $E_{12}$. The data output for this test case also appears in the technical report [7]. As can be seen (Figure 12(b)), the system is verified as unsafe. Moreover, the system has identified the offending location in the timed automaton. This state is drawn as darkened by the TACreator tool (Figure 12(b)).



(a) An incorrect TA



System verified as UNSAFE

(b) Incorrect TA failed

Figure 12: Example of an incorrect design

A foveate controller was modeled using TACreator 13(a). Each component of the controller was modeled as a state and data flow was modeled as an edge. Each component (Table 1) was benchmarked on randomly generated images to determine its timing constraint. Because a component passes data after processing is done, the model needed a measurement of elapsed time by which a message must be sent. This was done using two clock variables. In each component time elapses for $C_2$

A foveate controller was constructed by wiring together an event listener pipeline that is modeled with timing constraints (Figure 13(a)). In this system, a rectangular region of the real image plane is cropped and treated as if it were the camera. This virtual camera can be panned or tilted by adjusting the location of the rectangle. This was done because low level access to the the motors in the P/T/Z cameras for the purpose of velocity control was not possible. The raw data output for this test case also appears the full technical report [7]. As can be seen (Figure 13(b)), the system is verified as safe.

The verified foveate controller is demonstrated (Figures 14, 15, and 16). In figure 14, the scene viewed by the camera initially has no moving objects. In figure 15, the author appears in the field of view of the virtual camera. His motion is detected by the background subtraction algorithm, his pixels are segmented and an error is computed between the center of the image plane in the virtual camera (cross-hairs) and the centroid of the moving object. The PID control law computes the appropriate servo command and the virtual camera pans and tilts the camera to bring the center of the image plane to the object centroid (Figure 15). Once the center of the image plane is coincident with the centroid of the moving object, error goes to zero and the system comes to rest (Figure 16).

By structuring control system software as an event generator pipeline, a foveate controller can be modeled as a timed automaton and its real-time constraints analyzed using formal verification. Given a verified system, each component can be run at its characteristic rate as a real-time task. Using a pipeline design, we are able to design, benchmark, and verify individual components of a real world system. While the systems presented here are relatively modest, the real benefit of automated verification happens for non-trivial control systems. In particular, how the components interact with respect to real-time timing guarantees becomes difficult as complexity grows.

## 8 Future Work

The pipeline design lent it self to methodical design and implementation of a computer vision controller. Next steps include code generation from system model. Systems generated from verified specifications can then be targeted for specific real-time OS run-times by encapsulating each component with the necessary code required to make them real-time tasks.

This investigation into applications of model checking has also opened many questions. Very quickly, in implementing TACreator we ran up against the curse of dimensionality. One area for future investigation is to apply AI techniques to the search problem. In the literature, there are examples where transitions in the timed automata are augmented with costs and search is treated as an optimization problem. In other research, the authors suggest bounding the explosion of states using iterative deepening $A^*$ (ID$A^*$). An interesting area for future research would be to investigate stochastic search. By this approach, at each node, a probabilistic choice is made to determine which transition to take during search. As we continue searching in the infinite transition graph, we encounter nodes $(s', succ(\phi, e))$. Statistics on transitions, $(s, s')$ in the timed automaton, can be kept and used to bias probabilities used in searching the infinite transition graph. In doing so, the goal is to learn and verify the notion of most likely execution traces. Over many execution runs, if we can verify likely execution traces and compute their probabilities, an approximation can be made that verifies a system is correct with some degree of probability. Moreover, we can also compute the probabilistic bounds on these likely execution traces.

## 9 Acknowledgements

Thanks and gratitude go out to Prof. Neil Immerman for his encouragement and support in allowing me to undertake this research. Most importantly, I thank him for opening my eyes to the interesting field of model checking and its applications.

## References

[1] Luca Aceto, Augusto Burgueño, and Kim G. Larsen. Model checking via reachability testing for timed automata. In Bernhard Steffen, editor,

*TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer-Verlag, 1998.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] Gerd Behrmann, Thomas Hune, and Frits W. Vaandrager. Distributing timed model checking - how the search order matters. In *Computer Aided Verification*, pages 216–231, 2000.

[4] Michael H. Cohen. Design principles for intelligent environments. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Madison, Wisconsin, July 1998.

[5] D. Dill. Timing assumptions and verification of finite-state concurrent systems, 1989.

[6] K. Dutton, S. Thompson, and B. Barraclough. *The Art of Control Engineering.* Addison Wesley, 1997.

[7] Witheld for review. Witheld for review. Technical report, Witheld for review, 2005.

[8] D.A. Forsyth and J. Ponce. *Computer Vision A Modern Approach.* Prentice Hall, 2003.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition.* Addision Wesley, 2000.

[10] O. Grumberg, D. Peled, and E. Clarke. Model checking. In *Model Checking*, Cambridge, Massachusetts, January 2000.

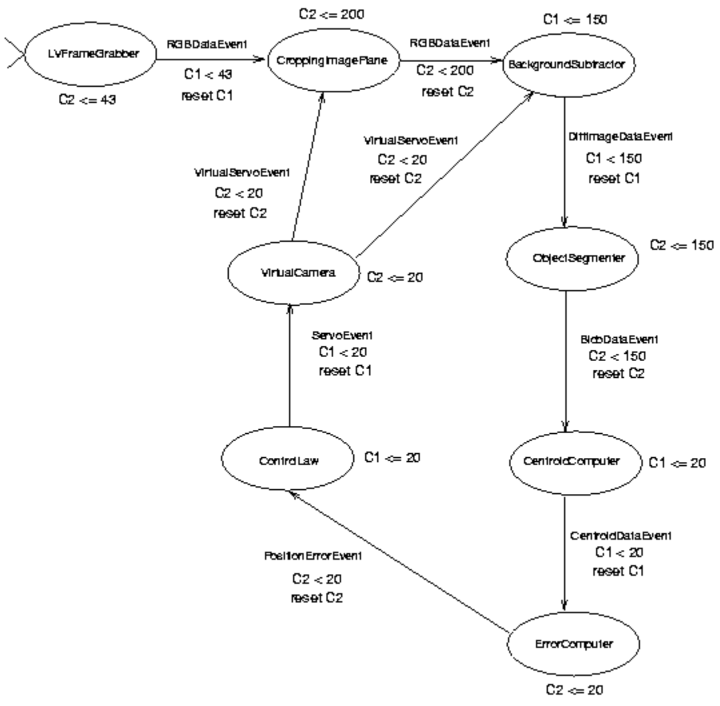
[11] J.D. Ullman J.E. Hopcroft, R. Motwani. *Introduction to Automata Theory, Languages, and Computation (2nd Ed).* Addison Wesley, 2000.

[12] J.A. Yorke K.T. Alligood, T.D. Sauer. *Chaos: An Introduction to Dynamical Systems.* Springer Verlag, 1996.

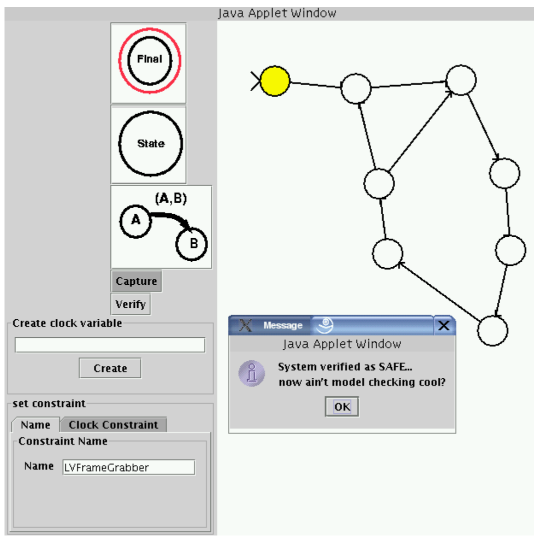
[13] J. Levine, T. Mason, and D. Brown. *Lex and Yacc.* O'Reilly and Associates, 1992.

[14] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (2nd edition).* Prentice Hall, 2003.

[15] S.H. Strogatz. *Nonlinear Dynamics and Chaos: With Applications to Physics, Biology, Chemistry and Engineering.* Perseus, 2001.



(a) Timing constraints on foveate controller



Name LVFrameGrabber

System verified as SAFE...
now ain't model checking cool?

(b) Foveate controller verified

Figure 13: Verifying a Foveate Controller

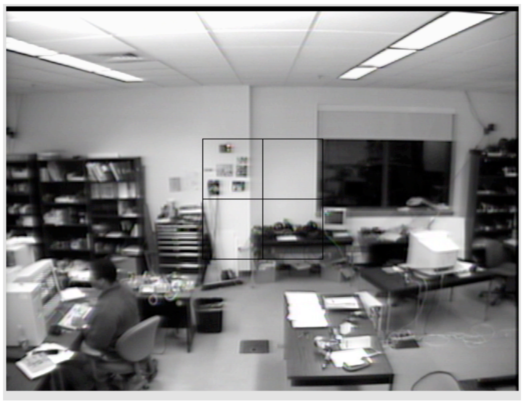


Figure 14: Initially no target

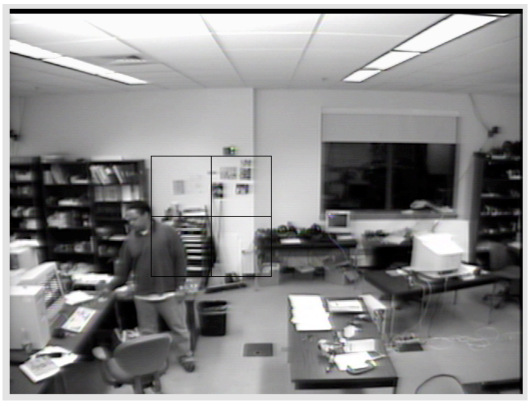


Figure 15: The author appears

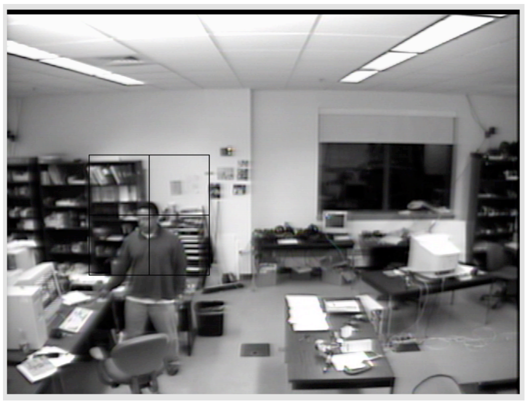


Figure 16: Foveated on the target