Visualizing Robot Behavior with Self-Generated Storyboards

Akkarit Sangpetch
Advisor: Professor David S. Touretzky

Senior Research Thesis
School of Computer Science, Carnegie Mellon University
May 10, 2005

Abstract

Achieving intelligent behavior on a mobile robot requires a mix of sensory processing, navigation, object manipulation, and human-robot interaction. Behavioral routines are often programmed as a collection of finite state machines, with states corresponding to actions, and state transitions triggered by asynchronous sensory events or timer expirations. Debugging these complex real-time behaviors or making them intelligible to a human observer can be challenging. Development of GUI tools can help programmers simplify this process.

In this project, we develop a graphical "storyboard" representation for visualizing a robot's behavior over time. Such a representation has uses beyond debugging. It can be used to make a robot's behavior comprehensible to its users. And it can provide a visual record of the robot's "performance" on a task, suitable for publication or display. The tool allows user to manipulate state machine diagram and automatically generate a storyboard view, based on layout of the state machine, from robot's execution traces. The tool is implemented in Java on a PC, and communicates with a Sony AIBO robot dog running the Tekkotsu application development framework developed at Carnegie Mellon.

Contents

1. Introduction

Problem description

Achieving intelligent behavior on a mobile robot requires a mix of sensory processing, navigation, object manipulation, and human-robot interaction. Behavioral routines are often programmed as a collection of finite state machines, with states corresponding to actions, and state transitions triggered by asynchronous sensory events or timer expirations. Debugging these complex real-time behaviors or making them intelligible to a human observer can be challenging. Development of GUI tools can help programmers simplify this process.

My Approach

This research project developed a graphical "story board" representation for visualizing a robot's behavior over time. Such a representation has uses beyond debugging. It can be used to make a robot's behavior comprehensible to its users. And it can provide a visual record of the robot's "performance" on a task, suitable for publication or display. The tool was implemented in Java as an Eclipse Rich Client Platform[1] on a PC, and communicates with a Sony AIBO robot dog running the Tekkotsu application development framework[2][3] developed at Carnegie Mellon.

To use my tool, the programmer first defines a graphical layout for the state machine. The layout tool connects to the AIBO via wireless Ethernet and downloads the state machine definition directly. The definition includes names for each node and each transition. The programmer then selects which nodes and transitions to display, and specifies their positions, sizes, and colors on the screen using the layout tool.

Once the layout has been defined, two kinds of graphical presentations are possible:

- A "state" display simply highlights those states that are currently active. (Tekkotsu's state machine framework supports a hierarchical organization and a fork operator, so multiple states can be activated simultaneously.) As the robot acts in the world, it switches from one state to another, and the highlighting changes accordingly.
- The more novel "storyboard display" uses a strip-chart format, with time increasing along the x axis. Active states are shown as polygons extending in time; their vertical positioning, color, and height match that of the corresponding node in the state graph. In addition, sensory events (such as button presses on the AIBO's body, timer expirations, or object detection events) are indicated by small icons along the top of the storyboard.

Storyboard creation is semi-automated. State activations are displayed automatically, but the programmer makes decisions about other kinds of supplementary information to log to the storyboard, based on what aspects of behavior he or she wishes to illustrate.

2. State machine visualization

Defining state machine using Tekkotsu Framework

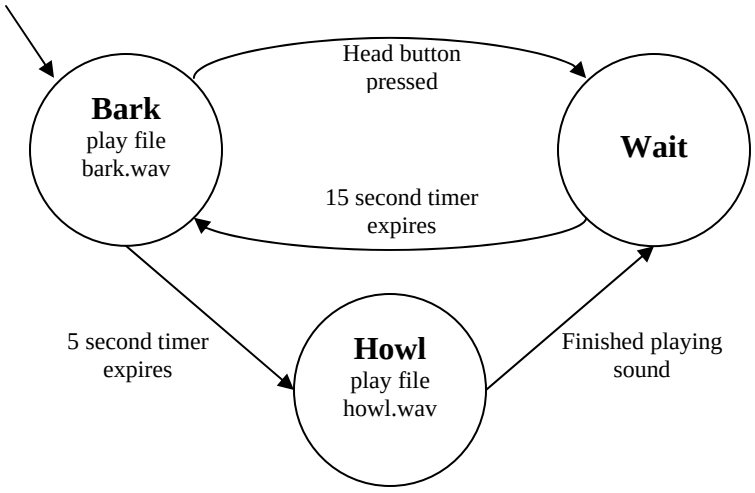
Tekkotsu supports state machines by providing `StateNode` and `Transition` classes, both are subclasses of `Behavior`.

A state is activated by calling its `DoStart()` function. This will in turn call `StateNode::DoStart()`, which will call the `DoStart()` functions of all the Transitions leading out of that state. Each Transition sets up one or more listeners for various types of events. If events of the appropriate types are received, the Transition "fires": it deactivates the source state by calling its `DoStop()` function, and then activates the destination state by calling its `DoStart()` function. Deactivating the source state causes the deactivation of all its outgoing Transitions (via their `DoStop()` functions), which causes them to remove their listeners.

To illustrate the concept, the code below defines a state machine for a simple Bark/Howl behavior shown in the diagram that follows:

Bark/Howl State Machine

```
01: #ifndef INCLUDED_SampleBehavior_h_
02: #define INCLUDED_SampleBehavior_h_
03: #include "Behaviors/StateNode.h"
04: #include "Behaviors/Nodes/SoundNode.h"
05: #include "Behaviors/Transitions/CompletionTrans.h"
06: #include "Behaviors/Transitions/EventTrans.h"
07: #include "Behaviors/Transitions/TimeOutTrans.h"
08: #include "Events/EventRouter.h"
09: class SampleBehavior : public StateNode {
10: protected:
11:  StateNode *startnode;
12: public:
13:  SampleBehavior() : StateNode("SampleBehavior"), startnode(NULL) {}
14:  virtual void setup() {
15:    StateNode::setup();
16:    SoundNode *bark_node = new SoundNode("bark","bark.wav");
17:    SoundNode *howl_node = new SoundNode("howl","howl.wav");
18:    StateNode *wait_node = new StateNode("wait");
19:    addNode(bark_node); addNode(howl_node); addNode(wait_node);
20:    EventTrans *btrans = new EventTrans(wait_node,EventBase::buttonEGID,
21:                                        RobotInfo::HeadFrButOffset,EventBase::activateETID);
22:    bark_node->addTransition(btrans);
23:    bark_node->addTransition(new TimeOutTrans(howl_node,5000));
24:    howl_node->addTransition(new CompletionTrans(wait_node));
25:    wait_node->addTransition(new TimeOutTrans(bark_node,15000));
26:    startnode = bark_node;
27:  }
28:  virtual void DoStart() {
29:    StateNode::DoStart();
30:    startnode->DoStart();
31:  }
32:  virtual void DoStop() {
33:    StateNode::DoStop();
34:  }
35: };
36: #endif
```



In this example, the state machine starts in state Bark, which has two outgoing transitions. One sets up a listener for the AIBO head button (`EventTrans`); the other starts a 5 second timer (`TimeoutTrans`). When activated, the Bark state plays a bark sound and then stays there. If the user presses the head button, the AIBO will transition to state Wait and the dog will wait for 15 more seconds before going back to the Bark state (and playing another bark sound.) Otherwise, if the AIBO remains idle for five seconds, the timer will expire and cause the AIBO to transition to the Howl state and play a howl sound. When the play finishes, the state will signal completion by throwing a status event, causing the dog to transition to the Wait state (`CompletionTrans`).

Tekkotsu's state machine structure is recursive; any node can contain an entire state machine within. As in the example above, the behavior is actually described as a state node named '`SampleBehavior`' whose setup function instantiated all nodes and transitions that make up the state machine. Then when `SampleBehavior::DoStart()` is called, it activates the state machine's start node (`Bark`) and starts the state machine.

Transport protocol

In order to get the state machine description from the dog, Tekkotsu provides a state machine spider which will collect information about the specified state machine and return the state machine model to the client on the PC in XML[5] format.

We use XML to encapsulate the state machine information. The information includes state node and transition identifiers, which can be specified by the user's code or automatically generated from class name by the framework. The content also provides C++ class names used for implementation of each state and transition. Each transition tag provides source and destination identifiers of state nodes. Note that it is possible to have multi-headed transitions with multiple source/destination nodes.

For example, our sample machine is represented with the following XML document:

```
<model>
  <state id="Bark" class="SoundNode" />
  <state id="Howl" class="SoundNode" />
  <state id="Wait" class="StateNode" />
  <transition id="btrans" class="EventTrans">
    <source>Bark</source>
    <destination>Wait</destination>
  </transition>
  <transition id="timeout5" class="TimeOutTrans">
    <source>Bark</source>
    <destination>Howl</destination>
  </transition>
  <transition id="timeout15" class="TimeOutTrans">
    <source>Wait</source>
    <destination>Bark</destination>
  </transition>
  <transition id="complete" class="CompletionTrans">
    <source>Howl</source>
    <destination>Wait</destination>
  </transition>
</model>
```

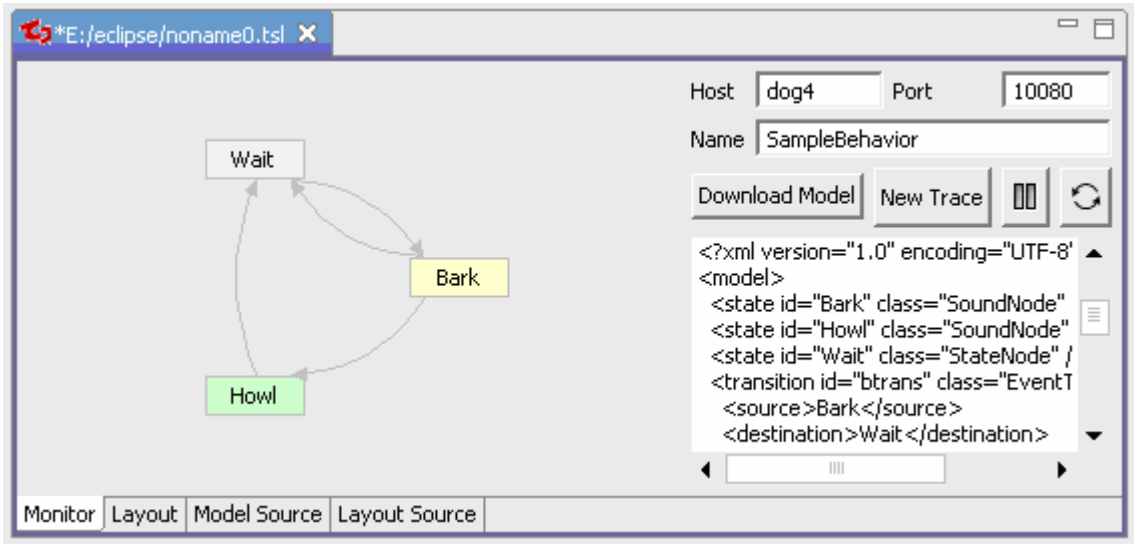
The complete document type definition is defined as follow:

```
<!DOCTYPE model [
 <!ELEMENT model (state*, transition*)>         // State machine description
 <!ELEMENT state (state*, transition*)>         // State node description
 <!ELEMENT transition (source+, dest+)>         // State transition description
 <!ELEMENT source (#PCDATA)>                    // Source node identifier
 <!ELEMENT destination (#PCDATA)>               // Destination node identifier
 <!ATTLIST state id CDATA #REQUIRED>            // State node identifier
 <!ATTLIST state class CDATA #REQUIRED>         // State node C++ class name
 <!ATTLIST transition id CDATA #REQUIRED>       // Transition identifier
 <!ATTLIST transition class CDATA #REQUIRED>    // Transition C++ class name
 ]>
```

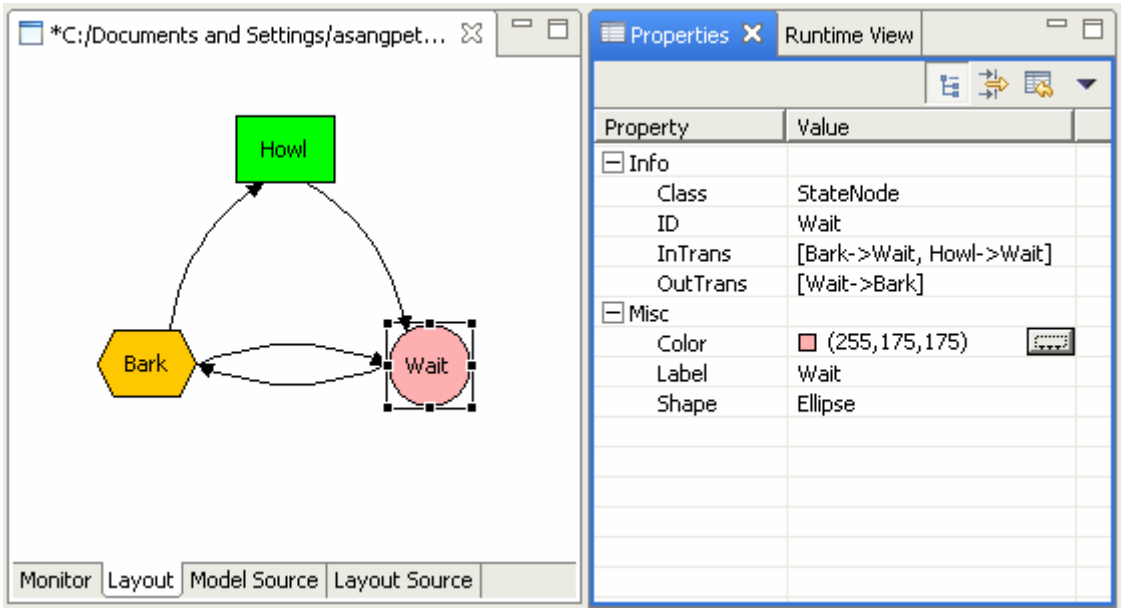
The retrieved information about the state machine model will be saved along with an automatically generated layout of the state machine. The user will be able to use the layout editor to further modify the layout and save it for future sessions.

State machine layout editor

A special state machine layout editor was implemented as part of this project. We use Eclipse's Graphical Editor Framework[4] to facilitate the development of this editor.
In order to use the layout editor, the user provides the hostname (or IP) of the AIBO, the spider server port and the name of the state machine. The client will request the state machine description from the AIBO and generate a simple layout by putting all state the state nodes in a circle and connecting all applicable transitions.



The user will then be able to manipulate the layout by moving or changing state nodes' shapes, sizes, colors and labels to suit their taste.



Once the layout has been defined, the user can save the layout and will be able to use it for monitoring sessions.

3. Self-generated storyboard

Transport protocol

The Tekkotsu framework also supports real-time event monitoring. The spider server on the dog can listen to various state machine events and generate event logs to the graphical client. The supported events are:

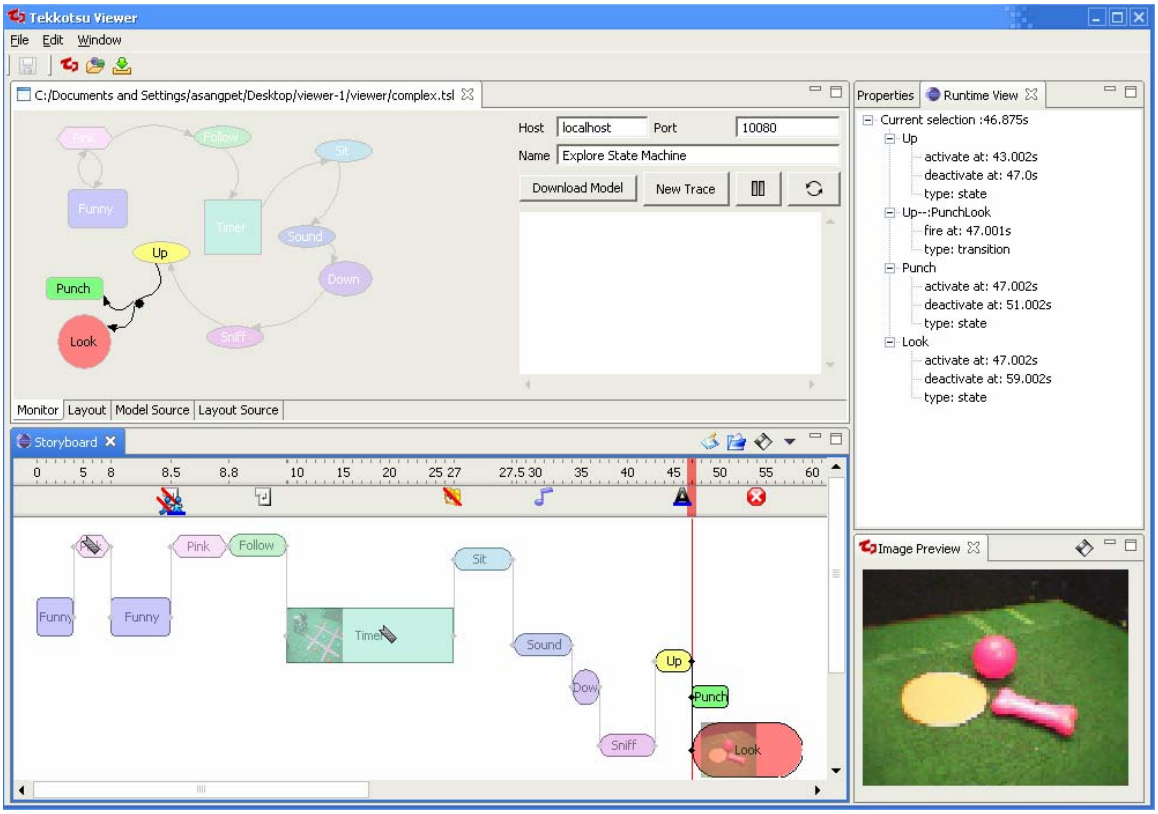
- State machine events: activation/deactivation/transition
- User-defined events: user can modify the state machine code to display custom message on the storyboard.
- System events: sensory events including button push, audio, vision sensor.
- Image events: taking image from AIBO camera or PC's webcam

We also use XML to encapsulate event logging, the document model is provided below:

```
<!DOCTYPE event [

<!ELEMENT event (#PCDATA|#CDATA|param*|(statestart|statestop|fire)*)> //general event tag
<!ELEMENT statestart (EMPTY)>                   // state machine activation
<!ELEMENT statestop (EMPTY)>                    // state machine deactivation
<!ELEMENT fire (EMPTY)>                         // a transition is fired
<!ELEMENT param (EMPTY)>                        // additional parameters

<!ATTLIST event type (transition|log|userlog|image|webcam) #REQUIRED> //type of events
<!ATTLIST event egid CDATA #IMPLIED>            // generator id (Tekkotsu)
<!ATTLIST event sid CDATA #REQUIRED>            // source id (Tekkotsu)
<!ATTLIST event etid (A|D|S) #IMPLIED>          // type id:Activate/Deactivate/Status
<!ATTLIST event time CDATA #REQUIRED>           // log time
<!ATTLIST event voff CDATA #IMPLIED>            // vertical offset relative to state node
<!ATTLIST event format CDATA #IMPLIED>          // image format

<!ATTLIST param name CDATA #REQUIRED>           // additional parameter name
<!ATTLIST param value CDATA #REQUIRED>          // additional parameter value

<!ATTLIST fire id CDATA #REQUIRED>              // transition id
<!ATTLIST fire time CDATA #REQUIRED>            // time at which the transition is fired
<!ATTLIST statestart id CDATA #REQUIRED>        // activated state node id
<!ATTLIST statestart time CDATA #REQUIRED>      // state node activation time
<!ATTLIST statestop id CDATA #REQUIRED>         // deactivated state node id
<!ATTLIST statestop time CDATA #REQUIRED>       // state node deactivation time

]>
```

What a storyboard looks like

The storyboard is displayed as a continuous timeline indicating state activation/deactivation. The appearance of the state nodes in storyboard is derived from the state machine layout including vertical position, color and label of each state node. The transition is displayed as a vertical line connecting the source and destination.



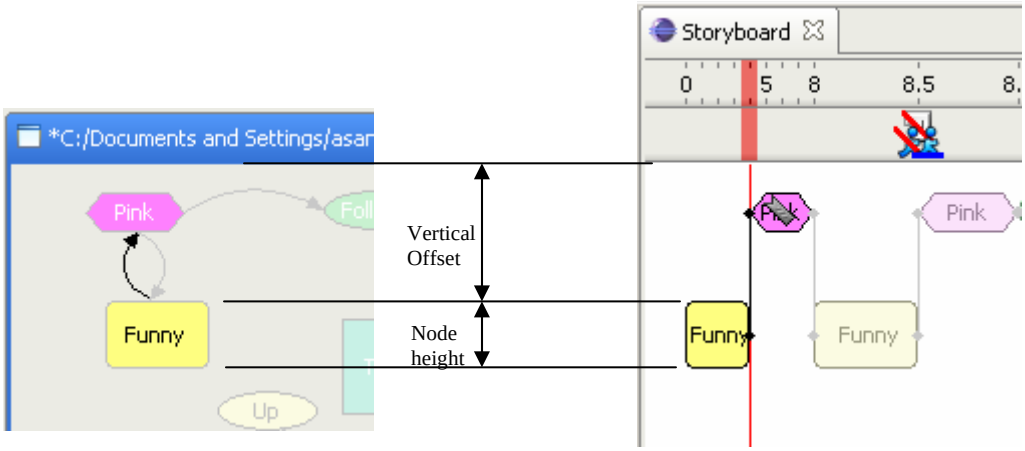
The figure above shows a sample monitoring session. The storyboard has a cursor which indicates the time at which the user wants to inspect. The state nodes/transitions which activate or deactivate at that time are highlighted in both the storyboard view and the state machine view. A textual description of the events occurring during that time is displayed in the runtime property view to the right. A custom user event is displayed as a little bookmark icon, which will show more information regarding the events if the user has the mouse pointer hover on the item.

The storyboard display is updated in real-time. After recording the execution trace, the user can save the current trace for further inspection or later presentation.

Storyboard layout

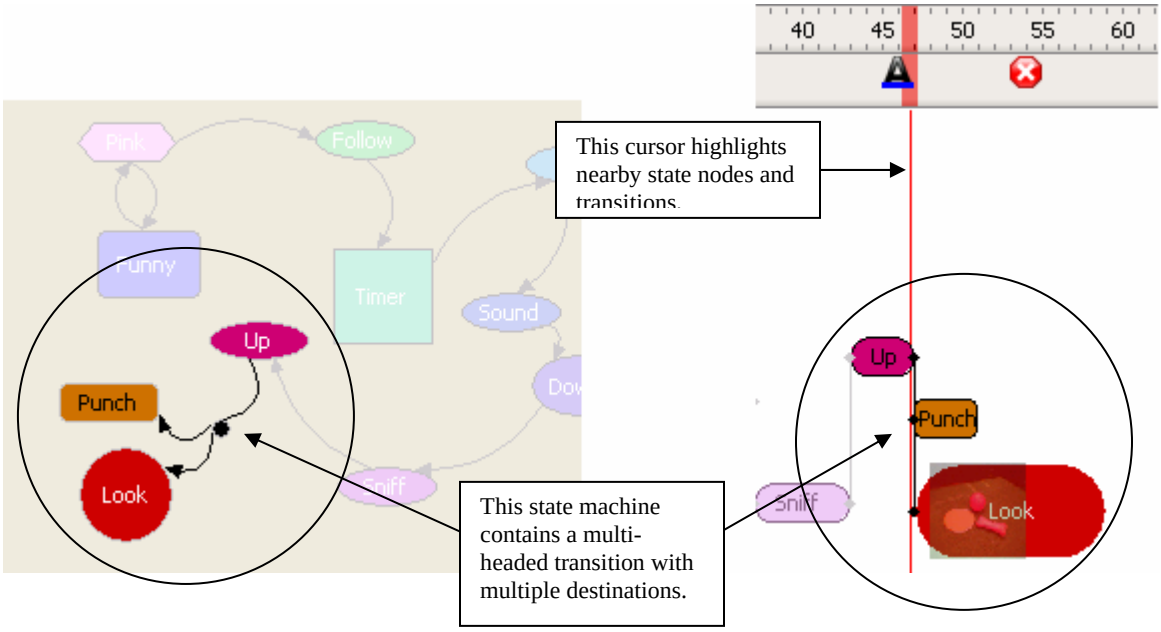
State nodes

The shape, height and vertical offset of state nodes in the storyboard utilize layout information from the state machine view. Horizontal offset and width of the chart indicates the node's activation time and period.



Transitions

Transitions are shown as vertical lines with small dots connecting sources and destinations. It is possible to have multi-headed transitions with multiple sources and destinations. In this case, the vertical transition line is stretched to connect all associated nodes. Sources appear to the left of the transition line; destinations appear to the right.

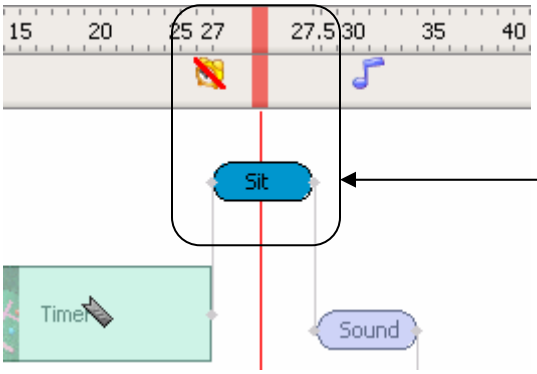


Cursor

In the storyboard view, the cursor is shown as a vertical red line stretching through the storyboard. This cursor indicates the time at which the user wants to inspect. Nearby nodes and transitions are highlighted and textual descriptions of these objects are shown in the runtime view panel.

Variable timeline

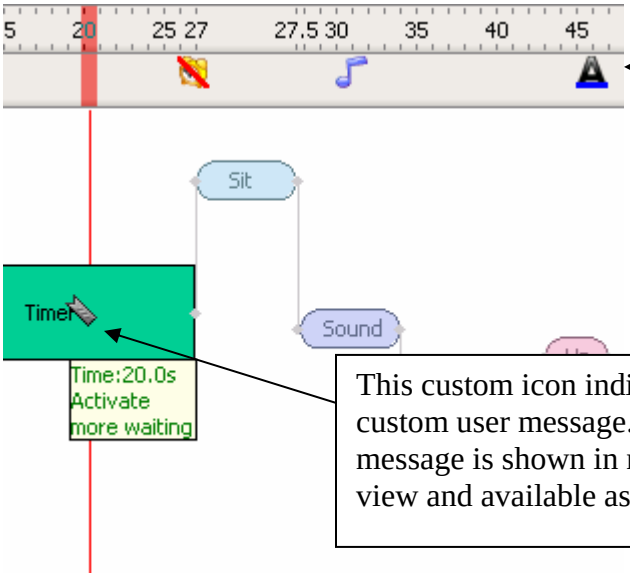
In order to display nodes with very short activation periods, which often confuse the user, the scale in the timeline is automatically adjusted to help users visualize these nodes.



This node is active for only 0.5 seconds. The timeline is stretched to display the node's width as defined by state machine layout instead.

Recording events

The tool allows the user to record and monitor other events beside the state machine related ones. These include user log, system event log, and image transfer from the AIBO.



This panel shows system event logs as icons indicating status of events (deactivate/activate/ periodical update).

This custom icon indicates a custom user message. The message is shown in runtime view and available as a tool tip.

Custom User Message

The tool generally requires no modification of user code to generate state machine events. However, users can manually insert a small piece of code into their state machine to display custom messages such as variable values, or parameters to facilitate the monitoring process. The user can also specify custom icons to be displayed on the storyboard by calling the following method:

```
logThis(“Hello world”, “hello.ico”);
```

System event log

The tool is also able to display system events as small icons (independent of the state node) in a system event bar, under the timeline of the storyboard. The user can either specify interesting events by declaring custom configuration during initialization of the state machine or use Tekkotsu’s Event Logger tool to determine which events should be logged onto the storyboard. The supported system events are:

- Button press and release events
- Timer events
- Motion command status
- Locomotion
- Audio starts/ends playback
- Emergency stop
- System text message

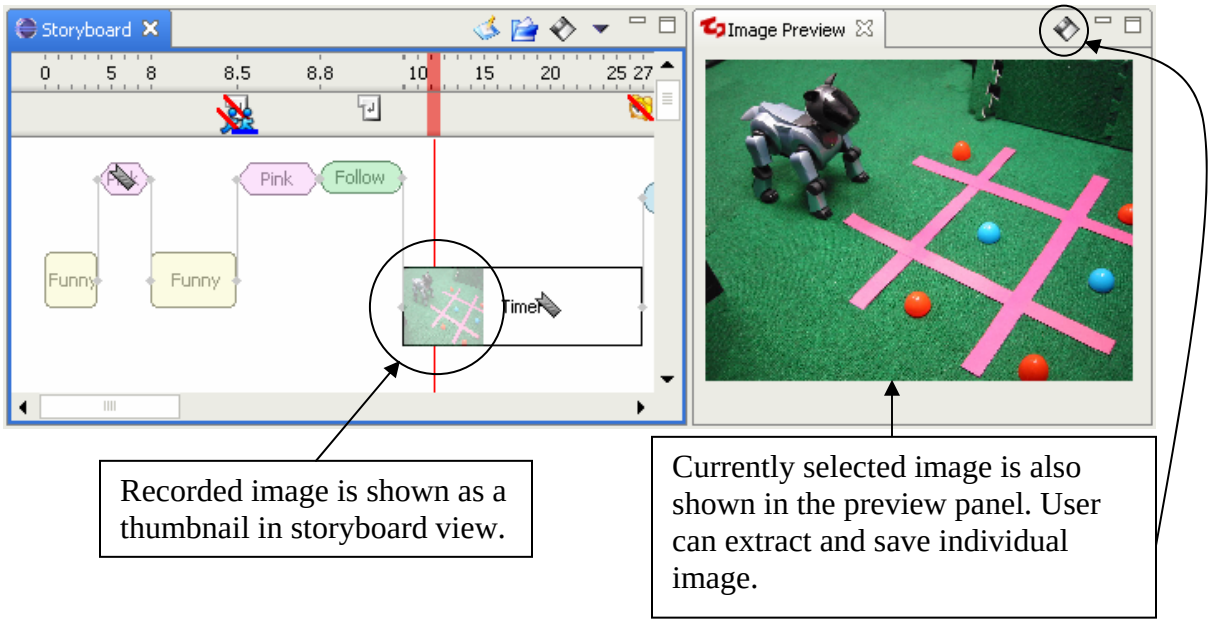
Capturing images from the AIBO or a webcam

The user can also insert code into their state machine commanding the AIBO to take a snapshot from its internal camera and send it to the graphical client. They can also connect the client PC to an external webcam and have the picture of the AIBO taken at a specified state using the following method:

```
logAIBOCam();
logWebCam();
```

The image will be taken (either from the dog or webcam) and stored on the client machine. An image thumbnail will be automatically generated and shown on the storyboard. If the user double-clicks on the thumbnail, the tool will show the full-size image in a separate window. A preview of the image is also shown in the image preview panel if the storyboard cursor is located on the thumbnail.

The binary data of images taken are stored as Base64 encoding integrated into XML file used for general event logging. Users can also extract individual image from event log file and save as jpeg format.



4. Evaluation and Future Work

User Testing

The tool was used in conjunction with the Tekkotsu framework in a robotics seminar course at State University of New York at Albany during Spring 2005. I was able to obtain user feedback regarding the interface and made various changes to the tool. For example, we include the automatic generation of the storyboard after users download model. The process assists new users to start the monitoring process without learning about the layout editor.

Future Work

The storyboard software created by this project will be tested by applying it to behaviors that other users build for the AIBO, such as a tic-tac-toe player currently under development. Upon completion, the software will be integrated into the Tekkotsu development framework, which is in use at a dozen universities around the world.

Various aspects of this project could also be improved. For example:

- The state machine editor could be improved by providing a right click context menu for users to change certain aspects of the layout.
- However, the storyboard view could benefit from transparency support so that multiple nodes could be stacked without obscuring the view. This will improve visualization for hierarchical state machines where it is possible to have a complete state machine inside a single state node of another machine. The current graphical library used by the project (Eclipse SWT[6]) does not support transparency. I attempted to switch to Java 2D graphical library to support transparency in storyboard view. However, the switch causes severe portability problems with Mac OS platform.

I also look forward to further improvement based on user feedback after the tool is publicly released.

References

[1] Jeff Gunther, Eclipse's Rich Client Platform, IBM developerWorks, July 2004.
[2] E.J. Tira-Thompson, "Tekkotsu: A rapid development framework for robotics," Master's thesis, Robotics Institute, Carnegie Mellon University, May 2004.
[3] (2004) Tekkotsu website. [Online]. Available: http://www.tekkotsu.org/
[4] Randy Hudson, Create and Eclipse-based application using the Graphical Editing Framework, IBM developerWorks, July 2003.
[5] T.Bray, J.Paoli, C.M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible Markup Language (XML) 1.0," W3C Recommendation, February 2004.
[6] Joe Winchester, "Graphics Context – Quick on the draw," Eclipse Corner Article, July 2003.