Graphical Animation of Behavior Models

Jeff Magee, Nat Pryce, Dimitra Giannakopoulou and Jeff Kramer
{jnm, np2, dg1, jk}@doc.ic.ac.uk
Department of Computing, Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, UK.

ABSTRACT
Graphical animation is a way of visualizing the behavior of design models. This visualization is of use in validating a design model against informally specified requirements and in interpreting the meaning and significance of analysis results in relation to the problem domain. In this paper we describe how behavior models specified by Labeled Transition Systems (LTS) can drive graphical animations. The semantic framework for the approach is based on Timed Automata. Animations are described by an XML document that is used to generate a set of JavaBeans. The elaborated JavaBeans perform the animation actions as directed by the LTS model.

Keywords
Labeled Transition System, Graphic Animation, Behavior Analysis

1 INTRODUCTION

A model-based design approach involves building analysis models early in the software lifecycle. These models can be developed shortly after the initial requirements capture and refined in parallel with further requirements elicitation so that early feedback on the operation of a proposed system can be fed back to customers and so that potential design problems are highlighted early. We have proposed such an approach, in relation to Software Architecture[1, 2], in which component behavior is modeled using Labeled Transitions Systems (LTS) and the overall behavior of a system can be formed by the parallel composition of these component models. We have developed the Labelled Transition System Analysis (LTSA) tool to support the approach.

The behavior of a model can be interactively explored using the LTSA tool. The output of such an execution is essentially a trace of action names. Each action is the abstract representation in the model of an input or output of the proposed system. In common with other model checking tools, the LTSA produces counter examples when it discovers safety property[3] or progress property[4] violations in a model. Again these counter-examples consist of action traces. A difficulty arises in interpreting the meaning of traces in relation to the original problem domain. Even when the meaning is clear to the model designer, the problem of communicating model behavior and the results of analysis to non-technical stakeholders of a system remains. Our motivation is thus to explore the value of graphic animation in validating behavioral models against requirements and in communicating the results of model analysis. The first step in this exploration, reported in this paper, is the development of suitable tools.

The idea of graphic animation is not in itself novel. For example, StateMate[5] supports animation through a set of predefined graphic widgets that display buttons, lights, dials and graphs. The novelty of the approach discussed here is the firm semantic foundation on which animations are constructed and the ease and flexibility with which an animation can be described and associated with the LTS model that drives it. The approach supports compositional animation development.

Section 2 of the paper describes the semantics of animation and its basis in Timed Automata. It shows how an animation is associated with an LTS. Section 3 describes how animations can be composed. Section 4 outlines the JavaBean based animation engine and how animations are generated from XML documents. Section 5 discusses related work and the paper concludes with an evaluation and discussion of the approach.

2 ANIMATION

We will use the example of a communication channel to illustrate our approach to animation. The channel takes an input message and either outputs the message or fails. The choice between outputting or failing is non-deterministic. The channel is modelled below as an FSP process. FSP[6] is the input notation for the LTSA tool. It is a simple process algebra used as a concise way to specify labeled transitions systems.

```
CHAN = (in -> out  -> CHAN
       |in -> fail -> CHAN
       ).
```

In the above, "->" denotes action prefix and "|" choice. The LTS that corresponds to CHAN is depicted in Fig 1.

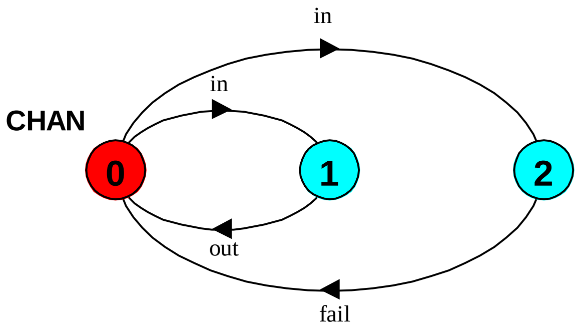
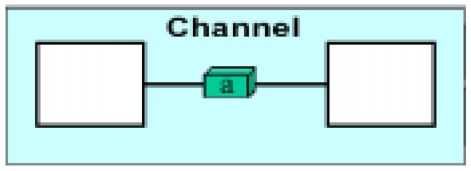
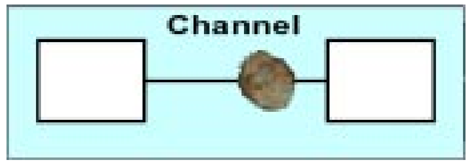


Figure 1 – CHAN LTS

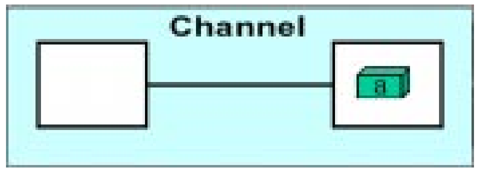
Three stages of a graphic animation of the channel are depicted in Fig 2.



(a)



(b)



(c)
Figure 2 – Channel Animation

Fig 2(a) depicts the situation in which an ***in*** operation has occurred and the ***out*** action has not yet happened. The message, represented by the block labeled ***a***, is moving from the input box on the left to output box on the right. Fig 2(b) depicts the state in which the channel ***fail*** action has occurred. This is animated by replacing the message block with an explosion. Fig 2(c) depicts the situation when the channel does not fail and the ***out*** action occurs. This simple animation consists of a single activity that moves the message block from the input box to the output box. Clearly the animation activity takes time. This time passes while the model is in a particular state. In the example, animation occurs when the model is in either state **1** or state **2**. That is the ***in*** action has occurred but the ***out*** or ***fail*** action has not yet occurred. In fact, *we abstract all animation activities by local clocks that measure the passage of time*. It was this observation that led us to the use of Timed Automata as the semantic basis for animation.

Timed Automata

Timed Automata [7] augment labeled transition systems with a finite set of (real-valued) clocks. Transitions are instantaneous and time can elapse in a state. A clock can be reset to zero simultaneously with any transition. At any instant, the reading of a clock equals the time elapsed since the last time it was reset. All clocks increase at a uniform rate, counting time with respect to a fixed global time frame. Clock constraints can be associated with transitions such that the transition can only occur if the current clock values satisfy the constraint. For a set $\boldsymbol{X}$ of clock variables, the set $\boldsymbol{\Phi(X)}$ of clock constraints $\boldsymbol{\varphi}$ is limited by the following grammar:

$$\varphi ::= \quad x \leq k \mid x \geq k \mid x < k \mid x > k \mid \varphi_1 \wedge \varphi_2$$

where x is a clock from the set $\boldsymbol{X}$ and k is a constant from the set of nonnegative rational numbers.

The timed automaton that describes the behavior of CHAN when it is combined with its animation is shown in Fig 3.

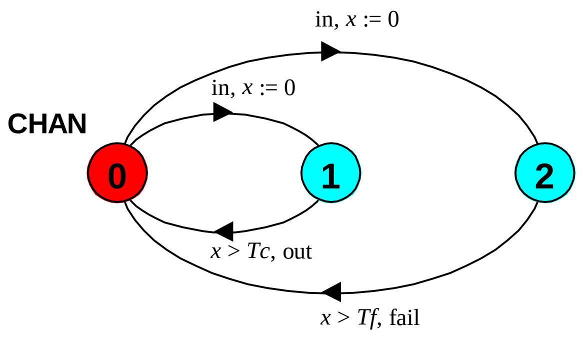


Figure 3 – CHAN Timed Automaton

Both of the transitions labeled by the action ***in*** reset the clock x that represents the channel animation activity. The ***out*** transition is only enabled when the clock value is greater than Tc. This represents the time at which the message reaches the box at the right of the animation display. The ***fail*** action is enabled after the clock value becomes greater than Tf. In the animation, Tf is less than Tc so that the explosion drawn by the fail action occurs before the box to the right is reached. The timed automaton describes the abstract behavior of the animated model. In the following, we outline how the concrete graphic display for the animation is associated with the LTS model in a way that is consistent with this abstract model.

Animation Activities

An animation is broken down into a set of activities. Each *animation activity* corresponds to a single clock variable in the timed automata model. The action of starting the activity corresponds to resetting the clock while the end of the activity corresponds to the satisfaction of a clock constraint of the form $x \geq k$. The progress of an animation activity corresponds exactly to the increasing value of a clock variable in the timed automata model. Each animation activity consists of a *command* to start the activity and one or more *conditions* that it signals as the animation progresses. In addition, the activity may provide commands to modify how the animation is drawn – in the example, the command to draw an explosion. The channel animation activity has the following commands and conditions:

commands:

`channel.begin`	`--` corresponds to $x := 0$
`explode`	

conditions:

`channel.end`	`--` corresponds to $x \geq Tc$
`channel.fail`	`--` corresponds to $x \geq Tf$

Combining Model with Animation

One of the desirable characteristics of a graphic animation facility is that it should not obscure or complicate the model specification. The description of how a particular model is animated should be separate from the model specification itself. This allows different animations to be applied to the same model and ensures that model development is not confused and complicated by the need to animate. We achieve this by defining relations that associate animation activity commands and conditions with action labels in the LTS.

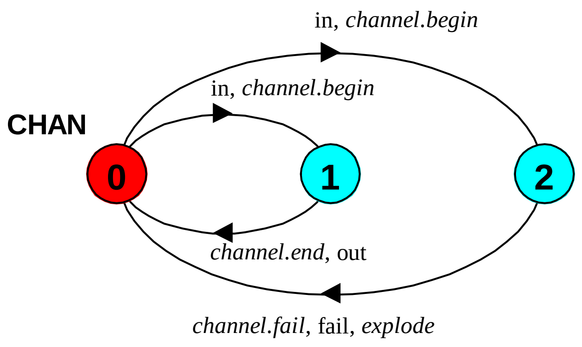


Figure 4 – `CHAN` annotated by animation `CHAN`

Animation activities are associated with model action labels using two relations: - ***Actions*** and ***Controls***. The *Actions* relation maps model action labels to animation commands and the *Controls* relation maps action labels to animation conditions. The relations for associating the channel animation with the CHAN model are declared in FSP as follows, where the infix "/" introduces a $\langle$*label*, *command*$\rangle$ or $\langle$*label*, *condition*$\rangle$ pair :

```
animation CHAN = "channel.xml"
   actions {
      in   / channel.begin,
      fail / explode
   }
   controls {
      out  / channel.end,
      fail / channel.fail
   }
```

An action label may be associated with both a command and a condition, e.g. *fail*. The effect of the animation declaration is to annotate an LTS with animation commands and conditions in the same way that timed automata annotate LTSs with clock resets and constraints. The animation above annotates `CHAN` as depicted in Fig 4.

Animation Execution

In the following, we outline the algorithm that executes an animation directed by an LTS. In the interests of clear exposition, we will ignore silent actions (τ) and the error state since these have little effect on the running of an animation. An error state or a state with no outgoing transitions terminates an execution.

We define an LTS P as a quadruple $\langle S, A, \Delta, q_0\rangle$ where:

- S is a finite set of states.
- A is a set of action labels known as the *alphabet* of P.
- $\Delta \subseteq S \times A \times S$, denotes a transition relation that maps from a state and an action onto one or more states,
- $q_0 \in S$ indicates the initial state of P.

The set of enabled actions E_q at some state q in the LTS is the set of actions that label outgoing transitions from that state:

- $E_q = \{a : A \mid \exists\, r \in S,\ (q, a, r) \in \Delta\}$.

An animation M is defined by the quadruple $\langle C, B, Actions, Controls\rangle$ where:

- C is a finite set of commands.
- B is a finite set of boolean conditions.
- $Actions \subseteq A \times C$ is the relation that maps an action to one or more commands.
- $Controls \subseteq A \times B$ is the relation that maps an action to one or more conditions.

An animation M applied to an LTS P partitions the alphabet of P into two sets, those that can be executed immediately and those that are controlled by animation conditions and thus must wait for these conditions to become true. These sets are:

- *Controlled* = **domain** *Controls.*
- *Immediate* = *A* – *Controlled.*

The predicate *Signaled(a)* is true for an action a if all of the conditions that it is mapped to are set to true.

NextState(q,a) computes the next state r in the LTS P such that $(q,a,r) \in \Delta$.

The following algorithm describes animation execution:

```
Animate P with M :
    curr := q_0
    loop
        while ($E_{curr} \cap Immediate \neq \{\}$ )
            choose a from ($E \cap Immediate$)
            execute { $c : C \mid (a, c) \in Commands$ }
            curr := NextState ( curr, a )
        end while
        if ($E_{curr} = \{\}$ ) exit
        while ({ $a : E_{curr} \mid Signaled(a)$ } = {}) wait
        choose ac from { $a : E_{curr} \mid Signaled(a)$ }
        execute { $c : C \mid (ac, c) \in Commands$ }
        curr := NextState (curr, ac )
    end loop
```

The animation algorithm ensures that all immediate actions, that are enabled, happen before the animation waits for controlled actions to be signaled. This is consistent with the maximal progress condition usually assumed for timed systems. We discuss later, how to check that a model/animation combination is free of Zeno behaviors – in our context, those behaviors involving an infinite number of immediate actions without time progressing.

Interacting with Animations

Animation execution as described above, once started, proceeds autonomously. Where more than one action is eligible as a choice, if the action is immediate and not controlled by an animation condition, the algorithm makes an arbitrary choice. This corresponds to letting the environment of the system make an arbitrary choice. Clearly we need to let the user interact with the animation so that these environment choices need not be arbitrary. To allow interaction, we introduce buttons which when pressed set conditions. These conditions control actions in the usual way.

Returning to the channel example, the animation once started runs continuously. The ***in*** action, which corresponds to the environment sending a message to the channel, happens autonomously. To allow the user to dictate when a message is to be sent, we introduce a ***send*** condition to control the ***in*** action as shown below:

```
animation CHAN = "channel.xml"
  actions { in   / channel.begin,
            fail / explode
          }
  controls{ out  / channel.end,
            fail / channel.fail,
            in   / send
          }
```

Introducing this new control mapping has two effects. It ensures that the ***in*** action cannot occur until ***send*** is set to true and it causes the animation engine to create a button. The button sets the ***send*** condition when it is pressed. The animation engine, by default, creates buttons for all those conditions named in the *Controls* relation that are not already used by animation activities. The channel animator together with the send button is depicted in Fig 5.

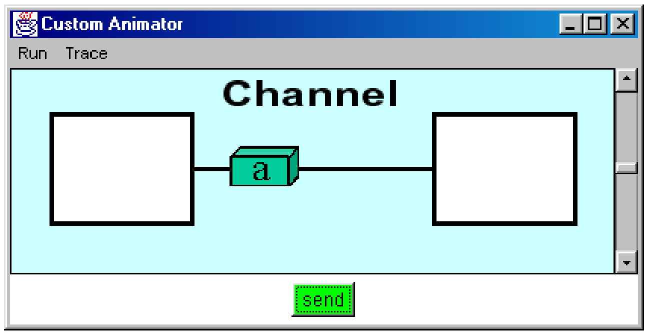


Figure 5 – Channel Animation with *send* button

The question that now arises is when to reset the condition set by a button? Normally an animation command resets all the conditions of the activity that it starts. However, there is no corresponding command for a button. Pragmatically, we have chosen to reset the condition associated with a button immediately after the button-enabled action occurs. All buttons are initially reset. The annotation of an LTS for a button ***go***, is shown in Figure 6, where ***~go*** is the command to reset the condition **go**.

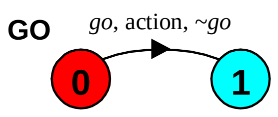


Figure 6 – LTS annotated with button condition

This scheme has the limitation that a button can only be associated with a single action, while the general case is that activity conditions can be associated with more than one action. However, our experience to-date has been that this limitation is not intrusive when designing animations.

Replaying traces

So far, we have described how animation is accomplished for an interactive execution of an LTS. The other use of animation is to replay traces generated as counter-examples during model safety and progress analysis. A trace is simply a sequence of actions. Replay of a trace *T* by an animation *M* is performed by the following algorithm:

```
Replay T with M :
    R := T
    repeat
        a := head(R ); R := tail(R )
        if ($a \in Controlled$) wait Signaled( a )
        execute { $c : C \mid (a, c) \in Commands$ }
    until ( $R \neq <>$ )
```

The algorithm simply executes actions in the sequence specified by the trace. However, where the action is controlled, replay waits until the controlling conditions are signaled.

Animation Behavior

The behavior of an LTS annotated by an animation – the traces it can generate or accept – is a subset of the behavior of the LTS without annotation. This becomes apparent if we reconsider the **Animate** algorithm. At any state, if there are both eligible immediate and controlled actions, the algorithm chooses immediate actions in preference to controlled actions. Controlled actions are only chosen if there are no eligible immediate actions. The controlled actions have lower priority. Consequently, to specify the behavior of an animated LTS, we use the low priority operator ">>" from[4], specified as follows:

For an LTS $P = \langle S, A, \Delta', q_0\rangle$ and set of actions $L \subseteq A$, the LTS in which the actions L are low priority is:

$P{>>}L = \langle S, A, \Delta, q_0\rangle$ where Δ is the smallest relation satisfying the rule:

$$\frac{P \xrightarrow{a} P'}{P >> L \xrightarrow{a} P' >> L} \text{ if } ((a \notin L) \text{ or } (\forall b \in (A-L),\ P \not\xrightarrow{b}))$$

The transition rule states that an eligible action in the prioritized system can be performed if it is not a member of the set of low priority actions L or there are no eligible higher priority actions in $(A-L)$.

With the low priority operator, we can now approximate the behavior of the LTS P animated by M with set of controlled actions *Controlled* as:

$$P >> Controlled.$$

This is an approximation that correctly constrains the relative ordering between immediate and controlled actions in the same way as the animation does. However, it permits more orderings of controlled actions than are permitted by the animation, since the animation schedules controlled actions with time. It is still a useful approximation since we can apply the progress check defined in [4], to check that a system is free of Zeno executions using:

`progress NONZENO` = { *Controlled* }

This asserts that in an infinite execution, it must be possible to execute one of the actions in the controlled set infinitely often. In practice, satisfaction of this property means that the animation will display some activity rather than freeze while a continuous loop of commands is executed.

3 COMPOSITION

Composition is of fundamental importance in our approach to developing models. Primitive components are modeled and analyzed before being combined into larger structures as dictated by the architecture of the target system[8]. In using animation as an aid to developing and validating models we require that it be of use in this compositional setting. This dictates that when we compose components, the animations associated with these components should also combine in a meaningful way. In the following, we show how the Timed Automata semantics that underpin animations provides a way to compose animations.

Timed Automata Composition

A timed automaton P_T is the tuple $\langle S, A, X, \Delta, q_0\rangle$ where:

- S is a finite set of states.
- A is a set of action labels known as the *alphabet* of P_T.
- X is a finite set of clocks.
- $\Delta \subseteq S \times A \times S \times 2^X \times \Phi(X)$, gives the set of transitions. $\langle s, a, s', \lambda, \varphi\rangle$ represents the transition from state s to state s' labeled by a with $\lambda \subseteq X$ the set of clocks to be reset and $\varphi \in \Phi(X)$ a clock constraint.
- $q_0 \in S$ indicates the initial state of P_T.

The composition of two timed automata:

$P_{T1} = \langle S_1, A_1, X_1, \Delta_1, q_1\rangle$ and $P_{T2} = \langle S_2, A_2, X_2, \Delta_2, q_2\rangle$ is

$P_{T1} \parallel P_{T2} = \langle S_1 \times S_2, A_1 \cup A_2, X_1 \cup X_2, \Delta, (q_1, q_2)\rangle$ where X_1 and X_2 are disjoint and Δ is defined by:

- for $a \in A_1 \cap A_2$, for every $\langle s_1, a, s_1', \lambda_1, \varphi_1\rangle$ in Δ_1 and $\langle s_2, a, s_2', \lambda_2, \varphi_2\rangle$ in Δ_2, Δ contains $\langle (s_1, s_2), a, (s_1', s_2'), \lambda_1 \cup \lambda_2, \varphi_1 \wedge \varphi_2\rangle$.
- for $a \in A_1 - A_2$, for every $\langle s, a, s', \lambda, \varphi\rangle$ in Δ_1 and every t in S_2, Δ contains $\langle (s, t), a, (s', t), \lambda, \varphi\rangle$.
- for $a \in A_2 - A_1$, for every $\langle s, a, s', \lambda, \varphi\rangle$ in Δ_2 and every t in S_1, Δ contains $\langle (t, s), a, (t, s'), \lambda, \varphi\rangle$.

Composition for timed automata is an extension of the normal LTS composition construction in that the transition for a shared action is annotated both with the union of clock resets for the transitions from each of the constituent automata, and with the conjunction of clock constraints.

Animation Composition

Remembering that in an animation, activities reify clocks and that clock resets are interpreted as activity commands and clock constraints as activity conditions, we can apply the timed automata composition construction to animations. An animation is defined by the set of commands C, the set of conditions B and the two relations *Actions* and *Controls* that map LTS action labels to commands and conditions, as discussed previously. The relations are used to annotate the LTS. To compose animations, we need to form the union of commands that label a shared action and the conjunction of conditions. This is done by forming the union of the *Actions* and *Controls* relations.

If animation $M_1 = \langle C_1, B_1, Actions_1, Controls_1\rangle$ and
animation $M_2 = \langle C_2, B_2, Actions_2, Controls_2\rangle$ then:

animation $M_1 \parallel M_1 = \langle C_1 \cup C_2, B_1 \cup B_2, Actions_1 \cup Actions_2, Controls_1 \cup Controls_2\rangle$

Where a model action maps to a set of animation commands, the *Animate* and *Replay* algorithms execute each of these commands. If it maps to a set of animation conditions, the algorithms require all of the conditions to be true before allowing the action to happen and any associated commands to be executed.

Example

To illustrate animation composition, we use a fragment from the Flexible Production Cell[9] animation depicted in Fig 7. This animation consists of activities to animate the operation of the input and output conveyors, the drilling and oven processes, and the crane that moves blanks between the conveyors and the manufacturing processes. In the following, the animation concerned with the crane is described.

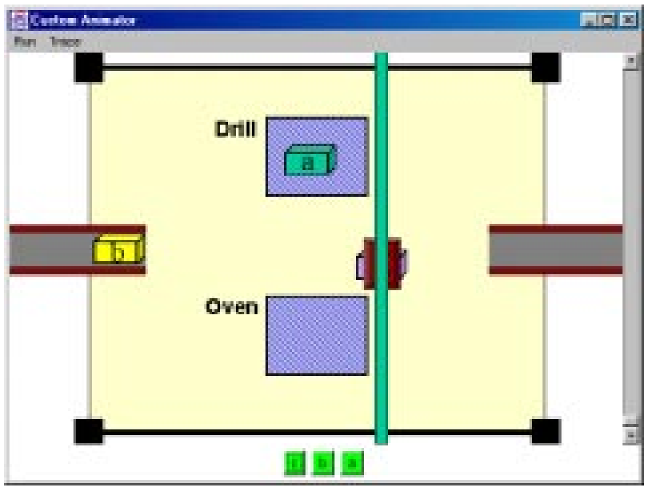


Figure 7 – Flexible Production Cell animation.

The crane is composed of two elements, the *gantry* that moves along the x-axis and the *head* that moves along the y-axis. The crane is positioned at a point (x, y) by a combination of gantry and head moves. The LTS of the gantry, annotated with its animation commands and conditions is depicted in Fig 8.

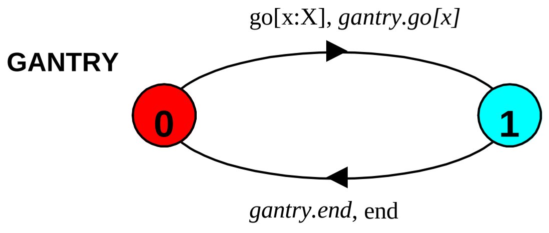


Figure 8 – `GANTRY` annotated with animation `GANTRY_ANIM`

The *FSP* description that generates this annotated LTS is given below:

```
range X = 0..4
GANTRY = (go[x:X] -> end -> GANTRY).
```

```
animation GANTRY_ANIM = "fmc.xml"
    actions  {go[x:X]/gantry.go[x]}
    controls {end/gantry.end}
```

There is no direct textual association between the animation and the process in the example. The association between a particular animation and the system it animates is made by user choice in the LTSA tool when the animation is activated. This gives the user the flexibility to associate different animations with a system and thus have multiple views of its behaviour. Different system models may also be associated at different times with the same animation.

The second element in the crane example is head movement. The annotated LTS is depicted in Fig 9.

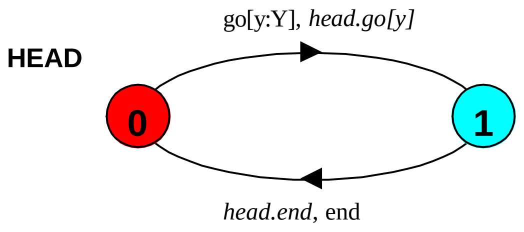


Figure 9 – `HEAD` annotated with animation `HEAD_ANIM`

The *FSP* description that generates this annotated LTS is given below:

```
range Y = 0..4
GANTRY =  (go[y:Y] -> end -> GANTRY).

animation HEAD_ANIM  = "fmc.xml"
    actions  {go[y:Y]/head.go[y]}
    controls {end/head.end}
```

We can now construct a process that coordinates gantry and head such that we can have composite actions that model moves to specific (x ,y) coordinates. Composing the head and gantry processes forms this crane control process:

```
 ||CRANE = ( GANTRY/{move[x:X][Y]/go[x]}
           ||HEAD  /{move[X][y:Y]/go[y]}
           ).
```

Composing the animations for gantry and head forms the animation for this composite process:

```
animation CRANE_ANIM  = "fmc.xml"
 compose { GANTRY_ANIM/{move[x:X][Y]/go[x]}
         ||HEAD_ANIM  /{move[X][y:Y]/go[y]}
         }
```

Note that the relabeling relations used in the process composition are the same as those used in animation composition to ensure the alphabets of process and animation remain consistent. The annotated LTS produced by applying the crane animation to the crane process is depicted in Fig 10. From this figure, it can be seen that a move action to a particular location starts both the gantry and head animation activities. The end action cannot occur before both of these activities have terminated.

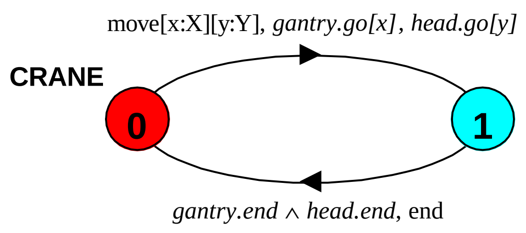

move[x:X][y:Y], *gantry.go[x]*, *head.go[y]*
CRANE
0
1
gantry.end ∧ *head.end*, end

Figure 10 – `CRANE` annotated with animation `CRANE_ANIM`

Animation compositionality allows the incremental development of model and animation. Components of a system can be modeled and animated before being combined into a larger system that has a composite animation by construction. In the following section, we outline the way that the graphic animation activities that are controlled by the model are specified and composed.

4 ANIMATION ENGINE - SCENEBEANS

Graphic animations are constructed from a library of JavaBeans that we have called *SceneBeans*. In the following, we first outline the architecture of *SceneBeans* and then show how a particular animation is specified using an XML document.

SceneBean Architecture

The basic entities in the SceneBeans architecture are *scene graphs*, *behaviors* and *animations*. A SceneBean animation communicates with an application such as the LTSA tool via *commands* and *events*.

Scene-graphs are a technique extensively used in 3D graphics, however here we apply them to 2D images. In SceneBeans, a *scene-graph* is implemented by a directed acyclic graph (DAG) of JavaBeans that draws a 2D image. Leaf nodes in the graph draw primitive shapes such as circles, ellipses, rectangles, and polygons. Intermediate nodes combine or transform their sub-graphs. Combination nodes either layer one sub-graph on top of another or choose one from a set of sub-graphs. Transform nodes apply an affine transformation to their sub-graphs – rotation, scaling, shearing or translation. Transforms may also change the way that their sub-graphs are rendered – for example, changing the color in which a node is drawn. Nodes in the scene graph expose one or more JavaBean properties by which their visual appearance can be modified. For example, a node that draws a circle exposes the radius of the circle as a bean property.

A *behavior* in SceneBeans is a bean that manages a time-varying value, announcing an event whenever the value changes. By connecting the events fired by a behavior to the property of a bean in the scene graph, the property can be made to change over time, so animating the visual appearance of the scene graph. A behavior updates its value from its initial parameter setting and from the passage of time. An animation thread that controls the frame rate of the overall animation signals the passage of time. Behaviors are started by a command and announce an event when they finish.

Our notion of an animation *activity* introduced in section 2 corresponds exactly to a *behavior* in the SceneBean framework. Each behavior maintains a clock that it uses to compute its output value. The behavior termination event is fired when this local clock value exceeds a maximum value set before the behavior was started. The animation thread updates the clocks of all behaviors synchronously. As a result, SceneBean behaviors respect the conventions for the local clocks of timed automata.

The times that are associated with an animation behavior in SceneBeans are in fact real times that are independent of the animation frame rate. The animation frame rate simply determines the smoothness of the animation. We can speed up or slow down the global time frame by making animation time faster or slower than real-time, however this does not affect the timing relationship between different animation activities, nor does it affect the frame rate.

The relationship between scene-graph, behaviors and animation thread is depicted schematically in Fig 11.

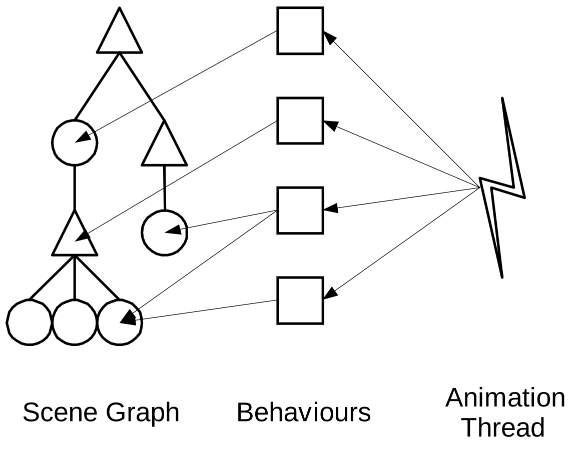


Figure 11 – SceneBean Architecture

A SceneBean *animation* encapsulates a scene-graph and the behaviors that animate the nodes of that graph. It acts as the manager for the behaviors encapsulated within it, routing commands and events. Most importantly, a SceneBean animation is also a scene-graph node, since this means we can compose animations, applying transformation and further animation as required. For example, in the earlier channel example, the explosion is a SceneBean animation that has been included as a node of the overall channel animation.

Specifying an animation in XML

To describe a specific animation, it is necessary to describe a scene-graph and the behaviors that animate that graph. A GUI-based tool that produces an XML document will eventually support animation design. However, although the eXtensible Markup Language (XML) [10] should most sensibly be considered as a machine-readable format, we currently specify animations directly in XML. To give a flavor of how this works, we present the encoding of the channel animation introduced in Section 2. The scene-graph and behavior for this animation are shown in Fig 12.

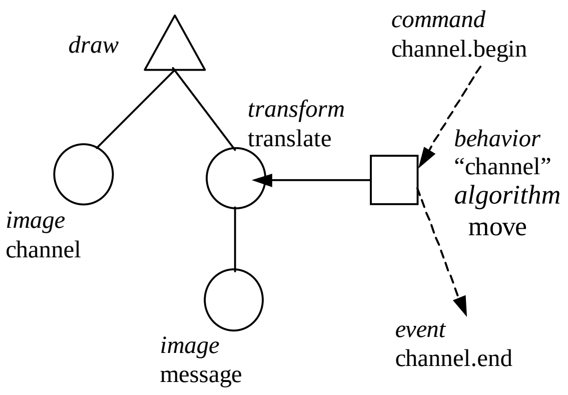


Figure 12 – Channel Animation Scene Graph

This generates the pictures of Fig 2. In the interest of brevity, we omit the elements that deal with channel failure and with displaying the explosion. The animation consist of a background image of the channel, consisting of the two boxes and the line between them, and the message image which is drawn on top of this background. The message image is moved by a translate transform fed by the behavior named "channel". This behavior is of type *move*, a behavior that over a period of time generates a set of values from a starting value to an end value. The XML file that describes the animation is listed below:

```
1  <?xml version="1.0"?>
2  <!DOCTYPE animation SYSTEM "scenebeans.dtd">
3  <animation width="400" height="136">
4  <behavior id="channel" algorithm="move"
5         event="channel.end">
6    <param name="from" value="71"/>
7    <param name="to"   value="323"/>
8    <param name="duration" value="2"/>
9  </behaviour>
10 <command name="channel.begin">
11   <announce event = "~channel.end"/>
12   <start behaviour= "channel"/>
13 </command>
14 <event object="channel" event="channel.end">
15   <announce event="channel.end"/>
16 </event>
17 <draw>
18   <transform type="translate">
19     <param name="y" value="64"/>
20     <animate param="x" behavior="channel"/>
21     <image src="image/message.gif"/>
22   </transform>
23   <image src="image/channel.gif"/>
24 </draw>
25 </animation>
```

This XML description is used to instantiate a set of Java Beans. The `<param>` tags translate into calls to bean property *set* methods. For example, line 6 translates to the call `setFrom(71)` on the behavior bean instantiated from `Move.class`. Line 20 adds the translate transform bean as a listener of the behavior bean for value change events. These value change events update the *x* parameter of the transform bean – resulting in movement of the message image.

We have chosen to completely define the interface between SceneBeans animation and an application, by a set of commands and a set of events. This simple architectural interface was chosen to facilitate integration of SceneBeans with applications other than the *LTSA* tool. However, it introduces a slight architectural mismatch between SceneBeans and the *LTSA*. As described previously, the *LTSA* considers an animation to be defined by commands and *conditions*. To turn events into conditions, we have introduced "not-events" that clear conditions in the *LTSA*. Line 11 announces `~channel.end`, the not-event for the condition `channel.end` signaled when the channel behavior terminates. The command that starts a SceneBean behavior must always send not-events for all events that it subsequently announces. The combination of not-event and event implements the condition state required by the *LTSA*.

Space has permitted us to describe only a very small subset of the comprehensive animation facilities supported by SceneBeans. The intension is to demonstrate that we have a concrete, powerful and flexible implementation of animation that is consistent with the abstract model assumed in sections 2 & 3.

5 RELATED WORK

Most verification tools provide the ability to execute the model specification as a way of simulating the system being modeled. The output of this simulation is displayed in the context of the specification. For example in SPIN[11], the simulator highlights statements in the Promela specification source as execution proceeds. The Concurrency Factory[12] displays the execution in the context of process diagrams specified in GCCS, a graphical notation for Milner's CCS[13]. UPPAAL[14], a tool based on timed automata, displays simulation results by high-lighting transitions and states of diagrammatic representations of automata produced using the Autograph tool[15]. Graphical animation in these tools thus refers to animation of some graphical representation of the model specification. This is clearly a useful facility in debugging and understanding models – it is a facility provided in the *LTSA* which animates LTS graphs – however, it does not address the problem of communicating in a domain specific way with requirements stakeholders unfamiliar with the modeling formalism.

Some initial work on domain specific visualization is reported by Heitmeyer[16] in the context of the SCR[17] simulator. They use the image of real instrument panels to display the outputs and controls for a simulation of the function of that control panel specified in SCR. The form of animation is similar in scope to that of the StateMate tool mentioned in the introduction. As far as we are aware, animation facilities, with the generality and flexibility of those described here, have not been applied to behavior modeling and verification tools.

Our work is perhaps closest in approach to work in the field of program visualization, although the objective of that work is to visualize program execution while ours is to target model visualization at a problem domain. A comprehensive account of the field of Software Visualization may be found in the eponymous book edited by Stasko et al[18]. We compare our approach with program visualization systems that represent two of the main approaches.

The Tango[19] and XTango[20] systems exemplify the first approach. In these systems, a program is annotated with "interesting events" that drive the visual animation. The animation is constructed using a "path-transition" paradigm. Interesting events correspond to our notion of *command* and paths are essentially a limited form of animation *behavior*. However, the notion of a *condition* that synchronizes program execution with animation is missing and perhaps not required since the systems are targeted at sequential algorithm visualization. Indeed as pointed out by Roman and Cox[21], Tango and related systems cannot easily be applied to concurrent programs.

The second approach exemplified by the Pavane[22] system as applied to Swarm[23] programs, visualizes program execution by declaring a mapping between program state and the visual representation of that state. Animation is thus a consequence of modifying the state. Our approach of providing a clear semantic framework for animation is predated by the declared intention of the authors of the Pavane system to put program visualization on firm formal foundations. Interestingly, our system may be considered to be a dual of Pavane, since Pavane depicts states and does animation during the transition between states. In our system, animation occurs while the model remains in a state and transitions can cause the elements in the picture to change instantaneously. We believe this difference results from the different objectives of the systems. Pavane is targeted at visualizing data parallel computations and as a result is state-based, while our system is targeted at depicting behavior and is event-based. The decoupling of animation from program/model is a common objective of both systems.

6 DISCUSSION & CONCLUSION

We have presented an approach to animating behavioral models that preserves a clear separation between model and animation. Animation need not interfere with the process of model development, specification and analysis – it is treated as an annotation of the model. This separation permits an animation to be applied to different models and allows multiple animations to be applied to a particular model. Although we have not yet done so, multiple animations can be applied concurrently to a model by forming the union of their animation relations.

This flexibility is achieved by using Timed Automata as the abstract basis for animated models. Where Timed Automata add local clocks to standard labeled transition systems, animated models add activities that reify clocks as visual behavior. We have exploited the compositional semantics of Timed Automata to permit the compositional development of animated models. Separation between model and animation is achieved by defining an animation relation that annotates an LTS with animation activities in the same way that Timed Automata are annotated with clocks. This animation relation provides equivalent flexibility for event-based systems to the flexibility Pavane achieves in mapping states to visual representations.

However, strict adherence to the clock constraints allowed by Timed Automata, when combined with the fact that the animation relation annotates all transitions labeled by an action rather than a specific transition, does limit the range of possible animation behaviors. For example, in some animations, it is desirable that for different starting situations, the activity has a different end time – in other words, abstractedly k is not a constant in the constraint $x \geq k$. This relaxation does not invalidate the composition rules for animations, the ability to replay property violation traces or the prioritized approximation used to check progress properties. However, it means the Timed Automaton that represents the abstract behavior of an animated model can no longer be formed simply by replacing animation commands by clock resets and animation conditions by clock constraints. In using the animation facilities, we have also discovered that it is useful in some problem domains to reflect physical constraints, such as the fact that two objects cannot occupy the same space, in the animation rather than the behavior model. In essence, the model reflects system behavior while the animation provides the environmental constraints on that behavior Again this does not seem to affect the composition of animations but the overall abstract behavior. Investigation of these issues forms part of our current work on animation.

The SceneBeans animation engine provides a flexible general-purpose framework for implementing animations. This flexibility arises from the use of JavaBeans to package behavior and graphical entities and scene-graphs as the way of combining these entities into an animation. XML has proved a useful notation for describing the organization of a particular animation. The XML document type definition (DTD) constrains the way the different elements of an animation can be combined. The SceneBeans framework can be easily extended by the addition of behavior and graphics beans. Animations can be packaged in a reusable way as XML files that can then be included into larger animations. We hope to exploit this flexibility in an animation design tool that will have associated with it libraries of domain specific animations. An exciting future prospect is the use of three-dimensional animations viewed in three dimensions.

Currently, we are gaining experience in the use of animation in a number of areas. We are applying it to animating workflows in the area of complex distributed service provision[24], to the development of pedagogic examples and to modeling a sub-system of an air traffic control system. In future papers we hope to report on our conjecture that animation fulfils a useful role in communicating both the intent of a model and the analysis results from that model. The *LTSA* application and SceneBeans animation package are available from:

```
http://www-dse.doc.ic.ac.uk/concurrency/ltsa-v2/
```

Acknowledgements

We are grateful to the EPSRC for funding part of this work under grant GR/M24493, to British Telecomms for the motivation to look at workflow as a target for animation, and to the EU under grant C3DS for funding work on modeling workflow systems.

REFERENCES

[1] J. Magee, J. Kramer, and D. Giannakopoulou, “Behaviour Analysis of Software Architectures,” presented at 1st Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, USA, 22-24 February 1999.

[2] J. Magee, J. Kramer, and D. Giannakopoulou, “Analysing the Behaviour of Distributed Software Architectures: a Case Study,” presented at 5th IEEE Workshop on Future Trends of Distributed Computing Systems, Tunis, Tunisia, October 1997.

[3] S. C. Cheung and J. Kramer, “Checking Subsystem Safety Properties in Compositional Reachability Analysis,” presented at 18th International Conference on Software Engineering (ICSE'18), Berlin, Germany, March 1996.

[4] D. Giannakopoulou, J. Magee, and J. Kramer, “Checking Progress with Action Priority: Is it Fair?,” presented at 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'99), Toulouse, France, September 1999.

[5] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sheman, A. Shtul-Trauring, and M. Trakhtenbrot, “STATEMATE: A Working Environment for the Development of Complex Reactive Systems,” *IEEE Transactions on Software Engineering*, vol. 16, pp. 403-414, April 1990.

[6] J. Magee and J. Kramer, *Concurrency - State Models & Java Programs*. Chichester: John Wiley & Sons, 1999.

[7] R. Alur and D. L. Dill, “A theory of timed automata.,” *Theoretical Computer Science*, vol. 126, pp. 183-235, 1994.

[8] D. Giannakopoulou, J. Kramer, and S. C. Cheung, “Analysing the Behaviour of Distributed Systems using Tracta,” *Journal of Automated Software Engineering, special issue on Automated Analysis of Software*, vol. 6, pp. 7-35, January 1999.

[9] A. Lotzberger and R. Muhfeld, “Task Description of a Flexible Cell with Real Time Properties,” FZI, Karslruhe version 2.1, http://www.fzi.de/prost/projects/korsys/ 1996.

[10] T. Bray, J. paoli, and C. M. Sperberg-McQueen, “Extensible Markup Language,” World Wide Web Consortium http://www.w3.org/TR/1998/REC-xml-19980210 1998.

[11] G. J. Holzmann, “The Model Checker SPIN,” *IEEE Transactions on Software Engineering*, vol. 23, pp. 279-295, May 1997.

[12] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky, “The Concurrency Factory: A Development Environment for Concurrent Systems,” presented at 8th International Conference on Computer-Aided Verification (CAV'96), New Brunswick, NJ, USA, July/August 1996.

[13] R. Milner, *Calculus of Communicating Systems*, vol. 92: Springer-Verlag, 1980.

[14] K. G. Larsen, P. Pettersson, and W. Yi, “UPPAAL in a Nutshell,” *Springer International Journal on Software Tools for Technology Transfer*, vol. 1, pp. 134-152, 1997.

[15] V. Roy and R. de Simone, “Auto/Autograph,” in *Computer-Aided Verification*, R. Kurshan, Ed.: Kluwer Academic Publishers, 1993.

[16] C. Heitmeyer, C. Kirby, and B. Labaw, “The SCR method for Formally Specifying, Verifying and Validating requirements: Tool Support.,” presented at 19th International Conference on Software Engineering (ICSE'97), Boston, Massachussets, USA, May 1997.

[17] R. Bharadwaj and C. Heitmeyer, “Verifying SCR Requirements Specifications Using State Exploration,” presented at 1st ACM Sigplan Workshop on Automated Analysis of Software (AAS'97), Paris, France, January 1997.

[18] J. Stasko, J. Domingue, M. H. Brown, and B. A. Price, *Software Visualization*. Cambridge, Massachusetts: MIT Press, 1998.

[19] J. Stasko, “TANGO: A Framework and System for Algorithm Animation,” *IEEE Computer*, vol. 23, pp. 27-39, 1990.

[20] J. Stasko, “Animating Algorithms with XTango,” *SIGACT News*, vol. 23, pp. 67-71, February 1992.

[21] G. C. Roman and K. Cox, “A Taxonomy of Program Visualization Systems,” *IEEE Computer*, vol. 26, pp. 11-24, December 1993.

[22] G. C. Roman, K. Cox, C. Wilcox, and J. Plun, “Pavane: A System for Declarative Visualization of Concurrent Computations,” *Journal of Visual Languages and Computing*, vol. 3, pp. 161-193, January 1992.

[23] G. C. Roman and H. Cunningham, “Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency,” *IEEE Transactions on Software Engineering*, vol. 16, pp. 1361-1373, December 1990.

[24] C. Karamanolis, D. Giannakopoulou, J. Magee, and S. Wheater, “Modelling and Analysis of Workflkow Processes,” Imperial College, London TR 99/2, 1999.