37

DAVE: The Distributed Algorithm Visualization Engine

by

Sean W. McGinnis

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical [Computer] Science and Engineering

and Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 22, 1998

Author_____

Department of Electrical Engineering and Computer Science

May 17, 1997

Certified by_____

Stephen J. Garland

Thesis Supervisor

Accepted by_____

Arthur C. Smith

Chairman, Department Committee on Graduate Theses

Abstract: The Distributed Algorithm Visualization Engine, or DAVE, is a Master of Engineering Thesis supervised by Doctor Stephen Garland of the Laboratory for Computer Science at MIT. DAVE is a learning tool designed to aid in the study of distributed algorithms by providing a visual environment in which users can construct I/O Automata and view executions of these automata. The program is designed as a companion to Nancy Lynch's Distributed Algorithms, and the visual layout of the program is identical to the diagrams from that text. DAVE is written in Java, and is written to allow easy extension of the existing DAVE code. DAVE has been written and is available for use, though performance speed is becoming something of a concern as the program grows in complexity.

Contents:

Section I: Introduction

The field of distributed algorithms is an extremely complicated subject. Students are hard pressed to gain a working knowledge of the subject in a time frame as limited as one semester. There is an untold amount of material available to supplement an education in distributed algorithms, but very few tools. Any tools that could aid the student in studying and understanding distributed algorithms would be of great benefit to both the student and the instructor.

The Distributed Algorithm Visualization Engine (DAVE) is one such tool. DAVE is a software package designed to aid students in constructing and studying Input/Output Automata. DAVE, as a supplement to a course in Distributed Algorithms, could be a very effective learning tool in the study of distributed algorithms.

DAVE is, first and foremost, a graphical tool. The entire client area, or workspace, of DAVE is graphical in nature. When using DAVE, the user *graphically* creates and links together representations of processors and channels. When this process of creating, configuring, and linking is complete, the user has a complete graphical model of an I/O Automaton (I/OA).

The tool is not merely graphical. Using DAVE, the user can also view the course of an execution within the I/OA she has created with the system. DAVE allows the user to load an execution into the constructed representation of an I/OA, and then play back the course of the execution, watching the changes in the state of the I/OA and the activation of the actions that caused those changes.

DAVE is designed to be used in tandem with Professor Nancy Lynch's book Distributed Algorithms, a definitive text on the subject. The graphical appearance of the DAVE program will appear familiar to anyone who has skimmed the book, or even just looked at the pictures.

However, DAVE is not merely a tool for the student. We have several other purposes in mind as well. One of the most obvious applications for DAVE is as a front end to an IOA simulator. IOA is a formal programming language based on the pseudo-code used in Lynch. At

this time, no compiler or execution environment exists for this language. However, when a simulator becomes available for I/O automata written in IOA, DAVE is a natural candidate for both a front end and a back end to that simulator. To use such a simulator, the user would have to dictate how the automata are joined together and what algorithms are loaded into each process. This functionality is already present in DAVE. In addition, DAVE can also be used to view the simulator's output. Trying to understand the execution of an IOA by reading a series of states and transitions is frankly nightmarish. However, there would be no real need, since DAVE already contains the functionality necessary to play back an execution of a given automaton. Indeed, the structure of the DAVE program is such that a simulator written in Java could easily be integrated with the program, creating a seamless graphical IOA simulator.

DAVE is not merely for students of distributed algorithms. The ability to visualize an entire IOA and then watch its execution step by step can be useful to even the most experienced researcher of distributed algorithms. There is no bound on the complexity of the system that can be built with DAVE except the memory limitations of the machine on which it is running, and the amount of time it takes to execute commands on an automaton of a given complexity. With a small investment of time, a user can create a persistent on-line model of an I/O Automaton and share that model with colleagues.

With these purposes in mind, DAVE was designed for maximum freedom, usability, and extensibility. The DAVE interface is designed to be familiar, both as an application and as a representation of I/OA. The internal structure of DAVE also allows it to be easily extended. Lastly, as few assumptions as possible were made when implementing DAVE, allowing as much freedom to the user as possible. Since DAVE is a graphically oriented application, this is especially true in the appearance of the client area.

The remainder of this document is subdivided as follows: Section II provides a brief background for distributed algorithms, I/OA, and the IOA language; Section III explains the design of DAVE; Section IV details the implementation of the DAVE system; Section V provides

our results; and Section VI concludes with our observations and conclusions about the DAVE system.

Section II: Background

To understand DAVE, we must first understand I/O Automata, which provide a convenient mathematical formalization of distributed algorithms in an asynchronous system. To understand what an asynchronous system is, it is best to compare it to its simpler cousin, the synchronous system. In a synchronous system, a "global clock" runs all automata. They perform an action at each clock "tick", thereby assuring that all actions take place synchronously, hence the name. In an asynchronous system, there is no such clock. Any action could take place at any automaton at any time. When discussing executions of I/OA, we impose some order on the actions – namely, each action occurs entirely *before* some actions and *after* other actions. However, there are no constraints about the exact timing of these actions.

An I/OA is a simple state machine in which transitions are associated with certain actions. These actions fall into three categories – *input*, actions from the outside; *output*, actions to the outside; and *internal*, actions visible only to the automaton itself.

An I/OA consists of four components:

1.) A signature: These are the actions of the automata – input, output, and internal.

2.) A set of states: This is a (not necessarily finite) set of states for the automata.

3.) A set of start states: A subset of the above set of states that are designated as *start states*.

4.) Transitions: For any given state s and any given action p, there is a transition (s, p, s') if the action p is *enabled* in s (see below).

The signature is straightforward. For an action to be classified as internal, it can only be visible to the automaton itself. Both the internal and the output actions are under the control of the automaton. However, the input actions assumed to be controlled externally, and thus can

happen at any time. For any state s and any action p, if there is a transition (s, p, s'), and we say that p is *enabled* in s. All input actions are enabled all the time – I/OA are *input-enabled.* This means that the I/OA must always be ready for an input action – there is no way for the I/OA to "block" the action.

While this input-enabling restriction may sound inconvenient, there are distinct advantages to having this property. First, it forces us to consider oddly-timed inputs, not just the inputs that we are expecting. Second, this input-enabling restriction makes the model work out well. Without going into too much detail, it can be said that the input-enabling property makes it appropriate to use simple notions of external behavior.

There is a fifth portion of an I/OA – the *tasks* partition. The tasks can be thought of as a description of the "threads" or "tasks" of the automata. However, as DAVE doesn't deal with the tasks partition, this is not relevant to the application.

An *execution fragment* is a finite or infinite sequence of alternating states and actions such that if the sequence is finite, it must end with a state. For an execution fragment to become an execution, the fragment must:

a.) Begin in a start state.

b.) End in a state where no non-input actions are enabled.

c.) For an automaton A, the transition $(s_k, \pi_{k+1}, s_{k+1})$ must be a valid action for all $k \geq 0$.

To observe only the external actions of our I/OA, a *trace* is used instead of an execution. A trace is an execution without the internal actions or states of the automata.

Composition

Composition is used to build a representation of a complex system by composing automata representing smaller portions of the system. When a system is composed, actons that have the same name are connected. Thus, if an action p occurs, all actions composed with p will occur.

Of course, there are restrictions on the actions that can be composed and the way they can be composed. First, no internal action can be composed (since, of course, that would expose the internal action to the outside world). Second, two automata can not be composed unless the set of output actions of those automata are disjoint.

Once a set of automata is composed, the output actions of the component automata become output actions of the system, input actions that are not composed with output actions of the component automata become inputs to the system, et cetera. Likewise, the (start) states of the composition are the vectors of the (start) states of the component automata.

The IOA Language

In order to use I/O Automata in a computing environment, a formal language is needed. This language is IOA, a programming language closely based on the notation used by Nancy Lynch in her book "Distributed Algorithms". IOA also utilizes a formal specification language, but we will largely ignore this aspect of the language. The IOA language is defined in IOA: A Language for Specifying, Programming, and Verifying Distributed Systems, by Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri.

A typical IOA program might look like this:

```
automaton Adder
  signature
      input add(i, j:int)
      output result(k:int)
  states
      value: int
      ready: Bool := false
  transitions
      input add(i, j)
        eff value := i + j;
        ready := true;
      output result(k)
        pre k = value ∧ ready
        eff ready := false
```

The parts of this IOA program are:

1,) **The signature** -- The signature follows the format described above. Each action is declared with its type. This notation will be especially important to DAVE, when we are deciding which transitions can be composed.

2.) **The states** – The states of an IOA automaton encapsulate the state variables. There is a wide range of types that these variables can be defined as. Normally, this would be an area of great concern when designing DAVE. However, as we will see later, we have boiled all variable types down to a common currency that allows us to avoid the massive complexity of implementation surrounding variable types.

3.) **The transitions** – The transitions of an IOA describe the effects of the actions that it can perform. Again we see the type declaration on the action – input, output, or internal. Inside the declaration of the transition is the "body" of the transition – that is, the operations on the automaton's state variables that are performed when the transition occurs. These actions are written in a form called "precondition-effect code". The precondition (**`pre`**) is the boolean condition that must be met in order for the transition to be enabled. The effect (**`eff`**) will be performed if the precondition is true. This code is of little interest in the design of DAVE, and the its interpretation lies in the realm of simulators.

The field of I/O Automata, and distributed algorithms in general, is a very broad field, filling many books. However, this brief discussion should give us the background we need to understand the workings of the DAVE system.

Section III: Design

The list of goals for the DAVE design is a long one. This section explores this list and lays out the design for the DAVE system. Since the power of DAVE is in its user interface, the interface will also be defined in this section.

To implement DAVE correctly, specifications will be needed for the following elements of the program:

1.) A set of tools. DAVE must present the user with a set of tools that will make it quick and convenient to assemble a set of processes into the desired IOA. We must define this set of tools, their appearance, and their purpose.

2.) A design for the *user interface* (UI). This includes the appearance of the tools, the appearance of the working area, and the appearance of the DAVE palette.

3.) A design for the executions. One half of the functionality of DAVE is the ability to view and play back executions. Since no standard currently exists for the representation of IOA executions, we will propose one as a stop-gap measure until a more established design becomes available.

The User Interface

Since DAVE is a user-interface driven program, we will explore that aspect of the program first. Along the way, as we define the appearance of the tools available to the user, we will define these tools. In order to define the user interface of DAVE, we must define the program in three general areas.

1.) The appearance of the client area. When we speak of the client area, we are referring to the "workspace" of the program. The workspace is the area in which the visual representations of processes, channels, compositions, etc. appear, and where these representations can be analyzed, connected, and modified.

2.) The functionality of the client area. The client area must support a set of user actions, such as arranging and connecting automata. These actions and their consequences must be defined.

3.) The appearance of the menus and toolbars. While the appearance of the standard Windows program (of which DAVE is one) has been well established over the years, it is worth noting here.

4.) The purpose and UI for all available tools. Each tool has a specific set of behaviors. These behaviors must be defined. Since each tool is different, the interface the tool presents to the user must also be defined. This may be as simple as presenting a different cursor to the user, or as complicated as popping up an entire menu of options.

General Principles of the Design

However, exploring the details of the design, some general principles about this design must be established. These are the overall guiding principles of the design and implementation of DAVE.

One of the strongest principles DAVE adheres to is providing the user with as much freedom and power as possible. The programmer should never make assumptions about what the user would like from the program – rather, if it could possibly be a subject of contention, the user should be allowed to choose whatever behavior she desires. Throughout the design and implementation of DAVE, this philosophy of "maximum user freedom" is adhered to as much as possible.

Of course, freedom often comes at the prices of added complexity. Making assumptions about the user's wishes, while it may impairs the user's ability to do what she desires, may also simplify the interface to allow greater accessibility for the novice user. This is a design trade-off that must be considered at every step in the design. At certain points, power may have to be sacrificed for simplicity, but this decision will be justifirf when it becomes necessary to do so.

One easy way to achieve simplicity in the interface is to make the interface *familiar*. DAVE is fortunate in this respect, in that there are two established sources from which to draw. The primary source is *Distributed Algorithms* which, fortunately, has established a well-defined graphical language – the diagrams that are used throughout the text (see Figure 8.3 of Lynch). This language allows DAVE to express every aspect of any IOA that the user may wish to assemble or observe. The secondary source in Windows itself. For more than ten years, the

Microsoft Windows interface has been evolving. This evolution has established a standard, easily recognizable appearance that is inherent in every Windows program. This standard is apparent in the design of the Java AWT, and is also apparent in the design of DAVE. Since DAVE is designed primarily as a Windows program, and developed on a Windows platform, no changes to the established Windows UI style are warranted. Thus, the interface of DAVE is very much the same as that of any other "standard" Windows program, from the color scheme to the buttons. One aspect that is very different from any other Windows program, as seen below, is the client area.

The other design principles of DAVE are more general, applicable to most programs, and a little more disposable when the necessity arises. The first is speed. The speed of the program is especially at issue since DAVE is implemented in Java – not a programming language renowned for its high performance. If this is such an issue, why use Java? Quite simply, Java allows DAVE to reach the most users – with Java's ability to compile once and run anywhere, DAVE can be distributed to many, many more students. This is especially important in the university environment, where the available platforms may range from Macintoshes to Silicon Graphics workstations. There is nothing more frustrating for the user than not being able to use the program she wishes to on the most available workstation.

The Client Area

As stated in Section I, one of the primary goals of the DAVE project was to serve as a companion to Nancy Lynch's book, *Distributed Algorithms*. To that end, the client area of the DAVE program to resembles an actual diagram from the pages of that book. The text is the inspiration for the visual elements of the client area.

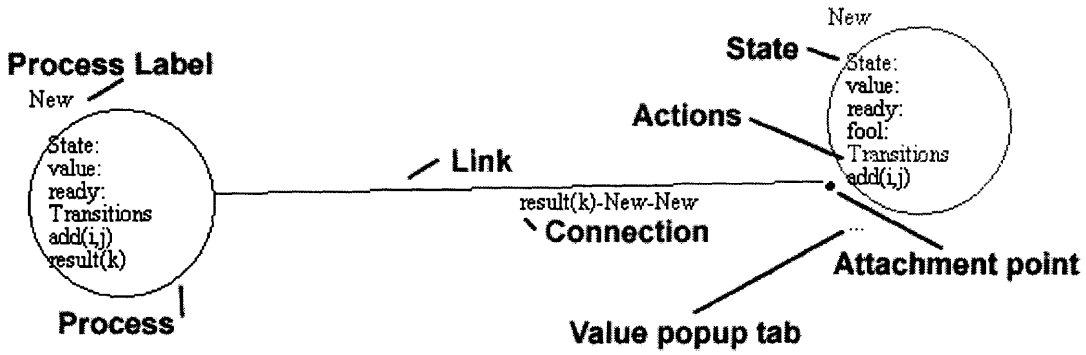

Figure 1: Detail from the client area of DAVE

Figure 1 is a detail from an actual screenshot of the DAVE workspace, with several key elements already in place on the page. Each element is explored in detail, its significance explained, why it is represented as it is, and what can be done with it.

1.) **The process** – the process is the basis for the rest of the program. The DAVE process represents an I/O Automaton as defined in Section 2. The purpose of the DAVE process representation is to allow the user to view the state and transitions of an automaton, its connection to other automaton, and the action in an execution that the automaton is taking. As such, the processes is designed to resemble, as much as possible, the representation found in Lynch. A process contains the automaton's *state* and *actions*. These will be explained in further detail below. The process has several visual elements that the user is able to change. These include the size, the color, the size and style of the text, its position on the screen, and its name. All of these characteristics are easily modified using the *configure* tool, described later.

2.) **The link** – The link represents a "conduit" between two processes. That is, if a composition of two actions is a "wire" between two processes, it would be a visual mess if DAVE represented every connection as a separate visual element. Instead, all of the wires are squeezed into one of two conduits. These conduits are established

based on which process contains the output action of a composition. Thus, between any two processes there can be at most two links – one from Process A to Process B, and one from process B to Process A. The link is a single visual element – an line ending with a large dot – representing all of the composed actions "from" one process "to" another. The purpose of the link is to allow the user to see the connections between processes. Since there can be at most two connections between any two processes, the client area of the program is significantly cleaner than if each composition was represented separately. This becomes crucial as the user creates more complex systems.

3.) **A connection/composition** – When two actions are composed, a "connection" is established. A connection is one of the "wires" discussed in number 2, above. As such, the connection has no visual representation except as a label on a link. When a connection is established where there is no link, DAVE will construct the link and add the composition. The composition's name will appear as the "label" to the link. Successive compositions will be added as additional labels to the link as time progresses. At this time, DAVE only allows the composition of two actions if one is an output action and the other an input action. This behavior does not correctly represent the model established in Lynch – DAVE must also allow the composition of two input actions.

4.) **A connection label** – Each connection in DAVE has a unique name based on the name of the actions being composed and the processes involved in the composition. The name of the action involved, complete with arguments, is the basis of the connection label. Then the names of the two processes involved, separated by hyphens, are appended to the end of the connection label, uniquely identifying the connection to both the user and other parts of the DAVE program.

5.) **The process label** – This is the name of the process. Since almost all I/O automata have a unique identifier (UID) state variable, it would be easiest if the process label could be used as this UID. To this end, all process labels must be unique. This greatly simplifies the task of creating connections. While this property does not necessarily dictate that the processes have UIDs (that is, this label need not be used in the actual execution of the automaton), it is still most convenient both for the user and the program if all of these identifiers are unique. A default identifier of "new" is applied to all automata. However, the process label is *not* the UID (that is, it is not a state variable), though it may be convenient to use it as such. Its real purpose is to uniquely identify the Java object that represents the process.

6.) **The attachment point** – The process label also serves as the attachment point for all of the connections made to the process. However, this is not set in stone. Below the process label is a small dot. By grabbing this dot and dragging it around the process, the user can set the point at which all links are connected to the process. This cleans up the appearance of the client area by leaps and bounds.

7.) **The state** – The state of a process is the list of all of the state variables declared in the algorithm for that process. This is represented simply as a list inside of the circle of the process. The name of a state variable is followed by a colon and its value. At this time, the value is always represented as text – DAVE has no support for non-text values such as images or sounds. Lists and arrays are represented as they are in Lynch. For values that are very long (that is, exceed the bounds of the circle) or are of a peculiar type (e.g. images, sounds) the value of the variable is replaced with a *value popup tab*. These are explained later. When the user views and execution of the automata, the color of state variables will change as they are affected by the course of an execution.

8.) **The actions** – The actions of the automata are usually not shown. This practice was adopted in order to save the valuable real estate inside of a process circle. The names of these actions will appear when they are called during an execution, and the names will present themselves in other ways when we are connecting automata. For the most part, however, these names are seen very rarely.

9.) **Value popup tabs** – The value popup tabs, or *hotspots*, appear in two cases. First, the tab will appear when a variable's value is either too long to fit inside of the process circle (e.g. a long list or array), or when a variable's value is of an unusual type (e.g. images, sounds, graphs, etc.). The second case is when the list of a process's state variables is too long to fit inside of the process circle. In the first case, left-clicking on the tab will bring up a window displaying the entire value of the variable. This window can be specialized by add-ons to display oddly typed values. In the second case, clicking on the tab will bring up a window presenting the entire list of a process's state variable inside of a scrollable text window. DAVE is responsible for deciding when a variable popup tab must be used.

The above list covers the appearance and meaning of the elements present in client area of DAVE. However, we must also define what actions the user can make inside of the client area in order to manipulate the processes and channels.

Commands within the Client Area

By left-clicking on a process, the user is able to *select* that process. When a process is selected, it is redrawn as a red-highlighted circle. The current process is the process that all executed commands (configures, connects) will be executed on. The current process is also the only process that can be moved.

By left clicking and dragging, the user can move the current process. This is often confusing to new users. To move an unselected process, the user must first select the process, and then left-click and drag in order to move it. To alleviate any confusion, a process is repainted with a red outline when it is selected as current. When a process is moved, any connections to the process also move accordingly.

By right clicking anywhere in the client area, the user brings up a popup menu containing process commands. These commands only apply to the current process. This shortcut saves the user the effort of moving between the client area and the toolbar in order to execute the most common commands on a process.

When the user wishes to connect two processes, she first selects the *source* process as the current process by left-clicking on that process. This is the process that has the output type of the action we wish to compose. The user then either clicks the connect button, selects connect from the process menu, or right clicks and selects connect from the process shortcut popup. When connect is selected, the cursor will turn into a targeting crosshair. At this point, the user clicks on the process that she wishes to connect to, the *destination* process. When this is done, a popup will appear displaying all of the actions that can be used to connect the source process and the destination process. By selecting one action from that list, the user is able to compose the selected actions and form a connection between the source and destination processes. At this time, two actions can only be composed if one is an output action and the other is an input action. In Lynch, however, two input actions can be composed. Support for this behavior is almost trivial to add to DAVE.

Finally, the user can not be restricted to the current viewport of the workspace. To solve this problem, the actual workspace much larger than the viewable area. There are two scrollbars that allow the user to scroll around the area. While this may seem to be an obstacle to some of the tools (for example, connecting two actions), in truth the format of the tools allows the user to

connect two processes effectively in spite of the fact that both processes involved may not be in the current viewport.

The Tools

For a user-oriented system such as DAVE, users must be provided with a wide range of tools in order to make their task as painless as possible. Assembling a complete, correct I/O Automaton can take a long time, even with the most convenient of tools. It is therefore incumbent upon the programmers to provide the most convenient of tools.

Creating an I/O Automaton representation

The first set of tools deals with constructing an I/O Automaton. These will be referred to as the *setup tools*. These are the tools related to creating, placing, configuring, and connecting automata. When discussing these tools, consider how to attach an algorithm to a process representation. Figure 2 shows the toolbar with the setup tools highlighted. Note the clustering of buttons on the right – these are the visualization tool buttons, discussed below.

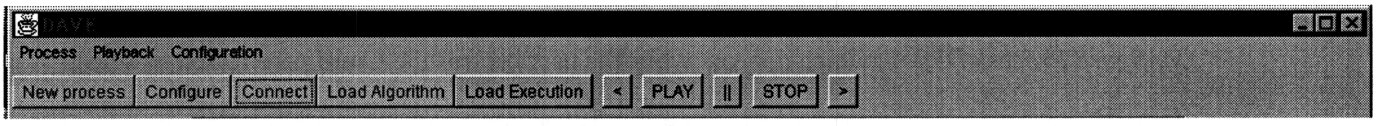

Figure 2: The DAVE Toolbar. The highlighted tools (the first four buttons from the left) are the ***setup tools****.*

The first of these tools, the **New Process** command, is used to *create a process*. This tool allows the user to create a new, blank process representation in the client area of the program. When this tool is executed, a new process representation appears in the center of the current

viewport. This new process has the default process label, "new", the default color and size, and is not defined (that is, has no state or actions). When DAVE was first being designed, a tool was considered that would create a default channel process, such as a FIFO message queue. This process would appear with a distinctive channel shape and have a default channel shape already loaded. However, once it was considered how many kinds of "common" automata there are, this tool was abandoned. Implementing one specialized automaton opens a bottomless pit of other specialized automata that could be implemented. While this may be a useful feature, it is beyond the scope of this project, and best left to the realm of future work on DAVE.

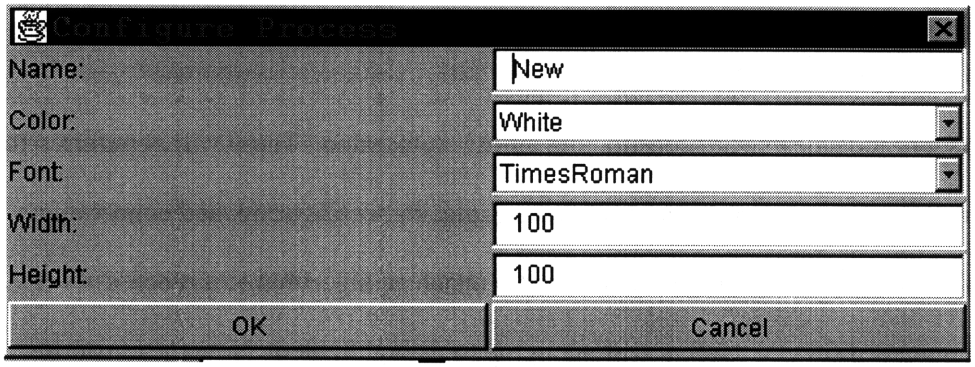

Figure 3: The "Configure Process" dialog

Once a process is created, the user may wish to configure it. Figure 3 is a picture of the *Configure Process Dialog*, which appears when the user presses the **Configure** button or selects **Configure** from the menu. The purpose of this tool is to allow the user to change the visual elements of the current process. The first on the list, and arguably the most important, is the *name*, or *process label*, as discussed above. The user can also change the color of the current process. This allows for cleaner looking diagrams – for example, all the processes in a diagram may be yellow and the channels green. The font can also be changed to serve the same purpose. The configure dialog also allows the user to change the *width* and *height* of the current process. In a system where most processes are circular, this may seem like an odd approach. It is not,

however, when the appearance of channels in Lynch is considered. In the text, channels appear not as circles, but as ovals, and it only makes sense to allow the user the ability to do the same in DAVE. Thus, the height and width fields allow the user to make any manner of oval she wishes, as well as change the size of the circle of a process. It is important to note that all values in the configure dialog are set to the current corresponding values in the current process. This gives the user some idea of what changes she wishes to make, and to what degree, particularly when addressing the size of the process.

Once the user has created a process and configured it as she wishes, she will want to load an algorithm into the process. The algorithm contains the state and actions of the automaton. These components are essential to compose an automaton with other automata and to use the visualization tools. The **load algorithm** tool allows the user to load an algorithm into a process. To do this, the user must press the *load algorithm* button. This command will load a selected algorithm into the current process representation. The command brings up a standard Windows file dialog, from which the user may select a *.alg* file. An example of this window appears in Figure 4. When the file is selected, it is parsed and loaded into the process. The text of the algorithm is stored within the process, and the process is permanently connected to that algorithm. *.alg* files are written in the formal IOA language as discussed in Section 2. Once the algorithm has been loaded, the state will appear inside the process, under the *State* heading. The variable names will appear, but no values will be attached. Values are only attached when the user loads an *execution* (see below).

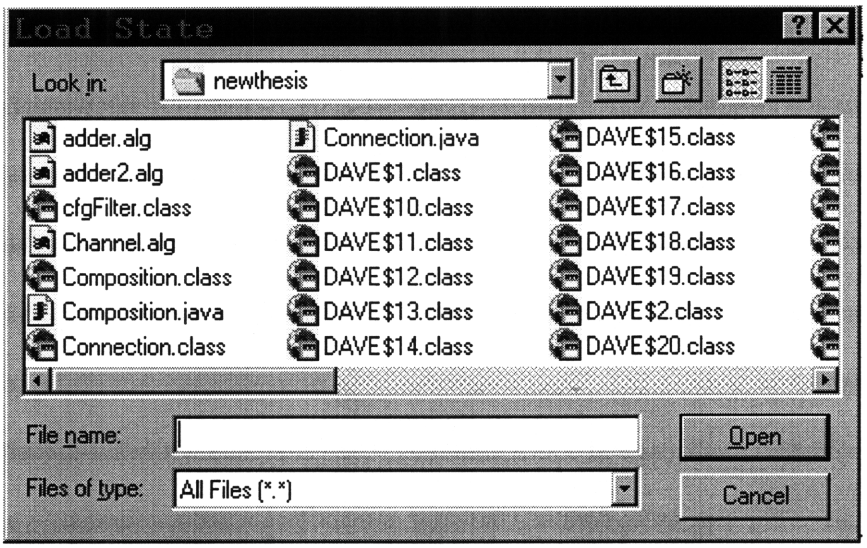

Figure 4: A typical DAVE file dialog.

When the user has two or more processes with loaded state, she may wish to *connect* the processes. Connection was discussed earlier in this section, but is explored here in more detail. First, what constitutes a *source* process? Quite simply, a source process has the *output* type of whatever action is to be composed. When the user selects a process as the *destination* process only those actions matching certain conditions will be in the list of available, connectable processes. These conditions are:

a.) The action must have the same proper name (that is, the text name of the action, not including the arguments).

b.) The action must be of the *input* type. A connection can only be formed if one process is of type *input* and one of type *output*. No actions with the same type can be composed, and internal actions can never be composed with any other action. This condition is not required for input actions in Lynch, and this condition will change in the next version of DAVE, as discussed above.

The way that a user composes automata in DAVE is not technically composition at all. In Lynch, one composes a *set* of compatible automata. All of the transitions with the same names in different automata are identified (connected). These identified transitions are related such that when one transition occurs, the other also occurs. In DAVE, the user identifies the transitions to be composed by hand. That is, the user does not compose entire automata, but instead connects two actions. The other actions within those automata – whether they have the same name or not – are unaffected. The end result of this connection is the same as composition – when one action occurs, the other also occurs. However, the user must keep in mind that this approach to connecting automata is not correct.

Now the user has assembled a set of processes, configured them, loaded the processes with the necessary algorithms, and connected them as she sees fit. What remains now is to *save the configuration*, either to work on later or to pass to a simulator. At present, there are two option for saving the configuration. Both are useful in different ways.

Saving and Loading Configurations

When the user has created, configured, and composed a set of automata in the DAVE client area that set of processes and connections is called a *configuration*. A configuration contains all of the visual information about the client area. These configurations can be saved and loaded by using the **Load Configuration** and **Save Configuration** tools. These tools are located in the **Configuration** menu. Since these commands are not as commonly used as the setup or visualization tools, they do not have buttons on the command bar.

The first way to save a configuration is to save the entire *DAVEDisplay* object (more on this object in Section 4), that is serialize the object and save it to a file. This makes it easy to load the file into DAVE again and, if a simulator is written with DAVE in mind, makes it easy for the author of that simulator to gain access to all of the structures already defined in DAVE. However, should the simulator be written independent of DAVE, this approach will never work.

It is impossible to extract the necessary information about the automata assembled in DAVE from the serialized object because the object is written to disk in a binary format specific to Java.

To counter this problem, DAVE has a second method of saving. Namely, DAVE *unparses* each process and connection and writes out the unparsed information to a text file. While this was much more technically challenging than the first approach, it has many benefits that the serialized object approach lacks. First, it is readable by any program – the text is simple for a programmer to read and understand, and the programmer can work from that information without knowing anything about DAVE. Second, the file is easily understood by the user. The user can peruse the file and understand it perfectly without running the DAVE program.

These second configurations are stored in *.cfg* file. A *.cfg* file might look like:

```
PROCESS:C1;270;49;100;100;java.awt.Color[r=0,g=0,b=255];java.awt.
Font[family=Dialog,name=Dialog,style=plain,size=12];Channel.alg;
PROCESS:P2;82;221;100;100;java.awt.Color[r=255,g=255,b=0];java.aw
t.Font[family=Dialog,name=Dialog,style=plain,size=12];Process.alg
;
PROCESS:C2;101;564;100;100;java.awt.Color[r=0,g=0,b=255];java.awt
.Font[family=Dialog,name=Dialog,style=plain,size=12];Channel.alg;
PROCESS:P3;533;584;150;150;java.awt.Color[r=255,g=255,b=0];java.a
wt.Font[family=TimesRoman,name=TimesRoman,style=plain,size=12];Pr
ocess.alg;
PROCESS:C3;600;341;100;100;java.awt.Color[r=0,g=0,b=255];java.awt
.Font[family=Dialog,name=Dialog,style=plain,size=12];Channel.alg;
PROCESS:P1;611;72;100;100;java.awt.Color[r=255,g=255,b=0];java.aw
t.Font[family=Dialog,name=Dialog,style=plain,size=12];Process.alg
;
PROCESS:User;79;880;100;100;java.awt.Color[r=255,g=255,b=255];jav
a.awt.Font[family=TimesRoman,name=TimesRoman,style=plain,size=12]
;user.alg;
COMPOSITION:receive(m)-C1-P2;
COMPOSITION:receive(m)-C1-P1;
COMPOSITION:send(m, i, j)-P2-C1;
COMPOSITION:send(m, i, j)-P2-C2;
COMPOSITION:receive(m)-C2-P2;
COMPOSITION:receive(m)-C2-P3;
COMPOSITION:send(m, i, j)-P3-C2;
COMPOSITION:send(m, i, j)-P3-C3;
COMPOSITION:leader(m, i)-P3-User;
COMPOSITION:receive(m)-C3-P3;
COMPOSITION:receive(m)-C3-P1;
COMPOSITION:send(m, i, j)-P1-C1;
COMPOSITION:send(m, i, j)-P1-C3;
```

The two types of statements in a .cfg file are the *PROCESS* statement and the *COMPOSITION* statement. By carefully examining each type, what information is stored in the file can be understood. The PROCESS statement is followed by a colon and the name or label of the process. This is followed by four integers representing the x and y coordinates of the process on the canvas, and the width and height of the ellipse representing the process. Following this information is the standard Java unparsing of the color of the process (in RGB values) and information about the font that the process is currently using. The last element in the PROCESS line is the .alg file attached to that process.

The COMPOSITION statement, by comparison, is very simple – it contains only the name. This is because all other information about the composition can be extracted from the name. From this name, the name of the action to be composed can be extracted. From the appended process labels, which processes are involved can be understood. And, from the text of the algorithms attached to the processes, it is apparent which actions are input and output for each process. Given this information, it is trivial to re-create the original connection.

Now that DAVE has all of this information, it is easy to implement the *Load Configuration* command. In this case, DAVE reads the unparsed or serialized *DAVEDisplay* object and presents the loaded configuration to the user. The file dialog displayed to the user is identical to the dialog displayed in the *Load Algorithm* command.

Now that the user has the tools to build her own I/O Automata, she will wish to utilize the second half of DAVE's functionality. Ultimately, DAVE will pass the saved configuration to a simulator, and the simulator will assemble an execution for the user. A text execution, however, would be hard to read at best. To solve this problem, DAVE allows the user to view the execution that is returned from the simulator.

Visualization Tools

The tools needed to load and view executions are called the *visualization tools*. One of the major design goals of this set of tools is to make them familiar and easy to use. For example, a user should not need to know how to use the setup tools in order to use the visualization tools of DAVE, allowing the most novice user access to these functions. Figure 5 shows the toolbar, with the visualization tools highlighted. Note how the buttons resemble the layout of a standard cassette deck, in an effort to make the toolbar seem as intuitive and familiar as possible.

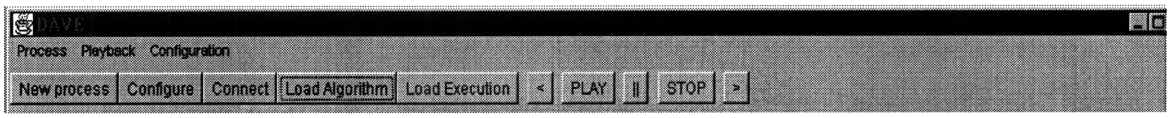

Figure 5: The DAVE toolbar. The highlighted buttons (the six rightmost buttons) are the ***visualization tools****.*

The first of these tools allows the user to *load an execution* – this is the **load execution** command. This command can be executed using the toolbar button or by selecting it from the **Playback** menu. This tool presents a file dialog identical to the one presented for *load configuration* and *load algorithm*. In this case, the user selects a *.ext* file. When this file is read, the state of the processes is loaded, and this new state is represented for the user in the client area of DAVE. The format of a *.ext* file will be discussed later in this document. Once the execution is loaded, all of the automata in the workspace should show a loaded state – the values assigned to each variable.

Now the real visualization tools come into play. The primary tool is the **play** tool. When the user presses the play button, the execution to plays out like a movie, showing what actions are being called and how the state variables are being effected. As the playback of the execution progresses, DAVE displays visually the changes that are being made to the I/O Automata

represented in the client area. When a variable has been changed by the execution, DAVE prints that variable in red, drawing the user's attention to the change. DAVE does the same thing with the compositions being called – when a composition is read from the execution, the composition is highlighted in red, as are the component actions in the processes involved. Figure 6 shows a process in the middle of an execution. Notice that the values for the variables have been filled in.

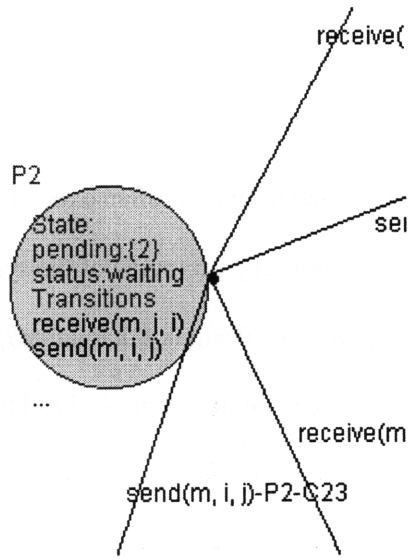

Figure 6: A process with its initial state loaded.

Similar behavior happens with the **step forward** and **step backward** commands. These commands are for closer examination of the execution. For example, the user may pause the execution somewhere in the middle when a particularly interesting action takes place. The user can then step forward or backward in single increments (that is, back up to the previous state, or move forward to the next). When the execution has reached its final state, the user can no longer step forward in the execution. At this point, she may **stop** or **step back** in the execution.

At any point in the execution, the user may wish to **pause** the playback. This stops the playing of the execution, but leaves the execution at its current state. This allows the user to

peruse the state of all the processes. When the user is ready to proceed with the playback, she need only press the play button again to proceed.

Often the user wishes to stop the execution altogether and back up to the beginning. For this purpose, there is the **stop** button. This serves much the same purpose as the pause button, but backs the execution back to its initial state. Note that all of the visualization buttons utilize the standard cassette player notations, again to make the interface as familiar to the user.

Executions

As mentioned above, no formal definition of an execution language existed before the DAVE project. In order to design a working prototype of the system, DAVE requires a language for this purpose. This section explores the design of that language.

The execution language is exceedingly simple and follows a few simple rules. It is easiest to explain by example:

```
STATE:
      P1:
            pending:={1}
            status:=waiting
      P2:
            pending:={2}
            status:=waiting
      P3:
            pending:={3}
            status:=waiting
      C1:
            buffer:={}
      C2:
            buffer:={}
      C3:
            buffer:={}
      User:
            mode := waiting
TRANSITION:
      send(1)-P1-C1
STATE:
      P1:
            pending:={1}
            status:=waiting
      P2:
            pending:={2}
            status:=waiting
      P3:
            pending:={3}
```

```
            status:=waiting
      C1:
            buffer:={1}
      C2:
            buffer:={}
      C3:
            buffer:={}
      User:
            mode := waiting
TRANSITION:
      receive(1)-C1-P2
```

This is a fragment of a .ext file, and is fairly self-explanatory. The STATE heading declares that what follows, up to the TRANSITION keyword, is a state declaration. The TRANSITION keyword declares that, until a STATE declaration, the text describes an action to be executed.

One of the most important things to notice in the above example is that every state is a complete state declaration – that is, every variable in every process is assigned a value, even if the value was not changed. This restriction allows much more freedom in the development of tools than a less restrictive approach. For example, a tool may extract a given state from the execution without running the execution. With this format, this would be very simple to do. However, if the state description at every step was not complete, to extract a given state would require playing out the execution from its beginning, or checkpointing the entire state of the automaton from time to time. This format also makes rewinding the execution much easier – the program need only step back one state in the text, rather than seeking repeatedly back through the text to assign values to every variable. However, because DAVE requires that the state be completely specified in the *.ext* file, the size of these files can be exceedingly large for complicated examples.

State is declared in a .ext file with the STATE: statement, followed by a newline. On subsequent lines, the user (or simulator) must declare the process name being affected, and the values of each variable above. In the initial state of the example above, notice the STATE statement, followed by a process name (in this case, P1). This is followed by a variable name and ":=", followed by the value and a newline. Since this state declaration is the first in the file, it is

called an "initial state". Every .ext file must start with an initial state.

All of the variables in this example, and in all other DAVE executions, have strings as values. In DAVE, the currency of values is strings – be the values integer, boolean, image, etc. This may seem either limiting or inconvenient, but there are a number of advantages of using strings. In the future, should new tools be added to DAVE, this established currency of strings will be easy to use, as the only functions necessary to interface with the existing *.ext* format will be parse and unparse features. There is also the distinct advantage that, for the majority of data types, the string is what would appear on screen.

The TRANSITION statements are somewhat more complicated than they appear. The naming convention when constructing compositions carries over into executions, and this name is again laden with information. The two process names appended to the end of the action name indicate to DAVE what processes are involved in this action. If the action is an internal action, there will only be one process name appended to the action name, indicating which process is executing the internal action. The proper name of the action indicates what action at each process is to be executed. However, the most important piece of information in the TRANSITION statement is the variable values placed inside the braces of the action, where formerly there were simply variable names. These assignments indicate to DAVE what arguments were passed to the transition, and allow the program to display these values in the connection, and at both the source and destination transitions.

Section IV: Implementation

The implementation of DAVE is a complicated system. The major goals of the implementation were to simplify the implementation and make it easily extensible for future programmers. This is especially important in this kind of program, which was developed ahead of other technologies that it is dependent on, such as simulators, or even the IOA language. To

ease integration with these future tools the program must be as simple and straightforward as possible.

Object-Oriented

To achieve this simplicity, the DAVE implementation adheres adamantly to the principles of object-oriented programming. In this system, where the theoretical objects are so clearly defined, it is simplest and best if the theoretical objects are translated into the actual objects in our code. Access to these objects is gained only through rigorous interfaces – there are no public instance variables. This allows pieces of code unrelated to an object to manipulate that object without inadvertently modifying other objects, with catastrophic results. If this discipline is obeyed, it makes the task of programming DAVE easier, and the tasks of future DAVE programmers.

One aspect of the program that lends a great deal to the simplicity of the implementation is the fact that the entire DAVE program is *event driven*. That is, once the main graphical elements of the program have been created, DAVE need only display these components, and then process the events that the user's interaction with these components generate. Using this event-driven model frees us from the complexities of dispatching commands, or attempting to run several threads of execution simultaneously within the program (actually, this isn't exactly true, as shown below).

Java

The choice of the programming language used to write DAVE has also kept the implementation simple and straightforward. DAVE is written entirely in the Java programming language, an object-oriented language very much like C++. However, where C++ comes with very little in the way of libraries, Java comes with a large *class library*, which gives programmers access to some of the most common constructs in programming.

One of the most powerful aspects of Java is the ability to "compile once, run anywhere". Java is an interpreted language. This allows the DAVE Java files to be compiled into a specialized representation called a *.class* file. These class files can then be used by a Java interpreter to run the program on any platform for which an interpreter exists. Actually, this is over-stating the power of the Java language. A common observation among programmers is that Java is "compile once, *debug* anywhere". Across different platforms, Java may behave differently than it does on the development platform.

Java was implemented with Windows in mind. One of the largest Java class libraries is the AWT (Abstract Window Toolkit), a collection of visual components that are common to Windows systems. These include windows, buttons, scrollbars, and a myriad of other components. In a program that is as graphics-driven as DAVE, these components are used quite often. Because Java provides many of the graphical components needed to implement DAVE, and the tools to create components that are not provided, the code can be kept fairly streamlined and simple.

The Java class library also provides much of the I/O functionality that DAVE requires. There are implementations of files, streams, and the different components necessary to use them efficiently. This includes file dialogs, filters, stream readers, and many other components. The use of these components can often be very complicated – all of these functions throw exceptions, and catching these exceptions can often be tricky. Several aspects of the Java I/O libraries make our programming DAVE easier, but several add new complexities to the task.

However, DAVE is not purely about graphics. Some of the most complicated aspects of the DAVE system are in the parsers and unparsers used throughout the system. Fortunately, the languages used by these parsers are designed very rigorously, or the existing languages used are very rigorous. However, these parsers will still be very complicated, no matter how simple the languages may be. These parsers will be explored later in this section.

The remainder of this section will discuss the intricacies of the DAVE code, the different objects involved, and how they interrelate. This section will also delve into the graphical aspect of DAVE, and the use of AWT components throughout the program. Finally, this section covers the algorithms used to parse the various file formats we use, and the unique structure of the playback tools.

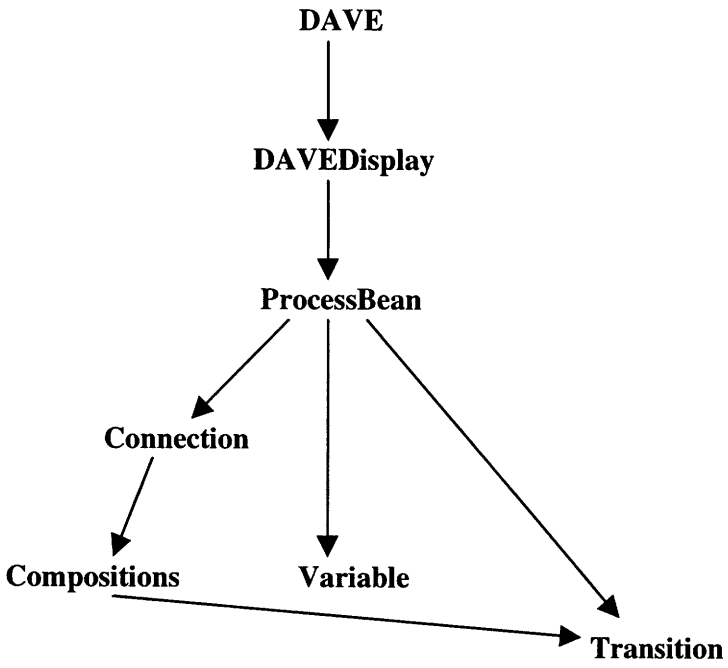

Figure 7: The DAVE object hierarchy.

Figure 7 shows an overview of the object hierarchy used in DAVE. Most functions in DAVE are distributed across several levels of this hierarchy. This section is broken down by functionality. That is, when a single function of the program is being discussed, the implementation of that function will be followed through every layer of the program.

The main function

The main function of the program is written in *DAVE.java* (see Appendix 2 for a complete code listing). This portion of the program actually has very little to do – it is relegated mostly to the function of event handling and dispatching, once the initial graphical components have been created. DAVE extends the Frame data type – this simply means that DAVE is an application window. The Frame data type provides functionality for such things as title bars, menus, and toolbars. All activity within the program takes place inside the DAVE window.

The *main* function of DAVE (which serves as the *main* function of the entire program) creates a DAVE object. The declaration of the DAVE type contains a long list of graphical components. As a matter of convenience, almost every graphical component used is listed in the beginning of this function, with few exceptions. Note the variable called *flow* of type *GridBagLayout*. This is a Java AWT layout component, and controls the arrangement of *Components* within the DAVE window. *flow* is broken into an irregular grid, and each component inserted into the grid takes up several blocks. The console buttons, for example, take up one block apiece, while the *DAVEDisplay* window (see below) takes up much, much more real estate.

One behavior that was difficult to implement in DAVE, but is an essential element of every Windows program, is the ability to *resize* the DAVE window. At this time, this behavior still does not work correctly in DAVE. The problem lies in the fact that the resize event is not being passed to the object representing the DAVE client area. Thus, when the application window is resized, the client area window is not resized. This leads to a visually unpleasing and inconvenient layout of the DAVE application window.

In the remainder of this function, the graphical components are created and put into the DAVE window. These components include buttons, menu items, scrollbars, and the client area object. Each component has associated with it an *ActionListener*. This is the foundation of our event-driven system – all actions taken by the program will stem from one of the functions in these *ActionListeners*. After all of the components have been attached to *ActionListener*s, they are added to the DAVE window with a certain set of *GridBagConstraints*, which dictates how the components will be placed in the *GridBagLayout*. Once this is done, the DAVE window is *pack*ed and *show*n, which finalizes the arrangement of components in the window and then displays that window. Notice that both the *main* function and the creation function for the DAVE object just end, rather than going into a command-processing loop. All other actions in DAVE are instigated by events.

DAVEDisplay

The *DAVEDisplay* type is also declared in *DAVE.java*. Almost all of the *ActionListener* functions outlined in the DAVE class merely dispatch calls to the *DAVEDisplay* object. It should be quite apparent that all of the power in DAVE lies in the *DAVEDisplay* object. The *DAVEDisplay* object does all of the event processing for the client area of the program. Since the *DAVEDisplay* has access to all of the *ProcessBean* objects (*ProcessBeans* are the object representation of processes), *DAVEDisplay* is also the intermediary between the DAVE object and any of the lower-level components in the program. The DAVE object contains one *DAVEDisplay* object as a member variable.

DAVEDisplay also has a number of graphical component objects as a part of its state – buttons, text boxes, pulldown menus. However, none of these components appear in the client area of the program. Instead, these components are placed in the various dialogs that the user will see as she uses the program. *DAVEDisplay* must have control of these components in order to extract the information it needs from them once the dialog to which the components belong has been dismissed.

The *DAVEDisplay* creation function is remarkably simple. The object need only set its parent and size, enable mouse events, and create a few member variables. Likewise, *getPreferredSize* is very simple, and really more of a technicality – the call is rarely used, but is necessary to implement *Component*, the superclass of *DAVEDisplay*.

Create a new process

All of the *ActionListener*s attached to the menu commands and console buttons call functions that implement the tools discussed in Section 3. The first of these functions is the **new_process()** function. This function finds the current position of the viewport and calculates the exact location of the center of the viewport on the underlying canvas. The function then sets

the default color and text of a new process, and calls the **new_process** function in the *DAVEDisplay* object *disp* with those arguments

This function implements a few subtle behaviors that should be noted. The first step is to create the new *ProcessBean* that will represent the new process. The arguments to this function are set to their default values in class *DAVE*, and the values passed into this function. The next step is to take the current bean and un-set the its current flag, and then make the new process the current process. It is assumed that every time the user creates a new process, she will wish to work on that process as the current one. This is a valid behavior, and also increases the familiarity of the interface. In most Windows programs, a newly created or opened object will be set as the current object, and there is no need to change this precedent. The new process is then added to the vector of processes that *DAVEDisplay* maintains, and the canvas is redrawn.

The new process representation is implemented by a class called *ProcessBean*. *ProcessBean* serves as an intermediary between the *DAVEDisplay*'s high level calls and the low level functionality of *Variable*s, *Transition*s, *Channel*s, and *Composition*s. Much of the graphical power of the client area is also present in *ProcessBean*. There are a lot of **PropertyChange** functions in the beginning of the class declaration. This is due to the fact that *ProcessBean* is a Java *Bean*. This functionality has little bearing on DAVE. However, it is significant that making *ProcessBean* a Bean also makes it a subclass of *Component*. *Component* is the base class for all of the user interface components in DAVE, from the buttons to the menu bar. As a subclass of *Component*, the *DAVEDisplay* has default behavior that allows it to process events easily. However, making *ProcessBean* a Bean may also have contributed to DAVE's performance problems (see Section 5).

In the case of *ProcessBean*, it will be necessary to become familiar with the variables of this class before proceeding. The most important variables are three vectors, **myState**,

myConnections, and **myTransitions**. These vectors will store the state variables (as *Variable* objects), connections (as *Connection* objects), and actions (as *Transition* objects) of the process, respectively. **myFont**, **myName**, and **myRect** contain the font, label, and bounding rectangle of the process. The other important variable is the *Rectangle* **hotspot**. This variable represents a rectangle bounding the screen space that corresponds to a hotspot. When the hotspot is pressed, a dialog displaying the entire state of the process is displayed.

Configure a process

The **configure** command, as discussed in Section 3, is used to set the visual properties of a process representation. The function **configure** of the *DAVEDisplay* object is deceptively simple – all of the configuration processing is actually done in the event handler for the return of the configure dialog, **actionPerformed**, which catches the command strings for the buttons in the configure dialog. When the "OK" button is pressed in the configure button, the dialog is hidden again, and the **actionPerformed** function extracts the required values from the graphical components that were contained within the dialog. These graphical components – the text boxes, pulldown menus, and labels – are all member objects of class *DAVEDisplay*. When the dialog box is hidden, the *DAVEDisplay* object extracts the values contained within these components and changes the current process accordingly. Notice also the **texttocolor** function – this allows **actionPerformed** to translate the color name from the configure dialog into an actual Java color. This function will pop up again when setting global color schemes.

configure, instead of calling a **configure** function lower in the hierarchy, calls the individual functions of *ProcessBean* used to set the graphical properties of the object. None of these functions is particularly interesting from an implementation standpoint – they merely set the variable that they correspond to to the value passed in as an argument.

One notable problem with DAVE is the speed at which the **configure** dialog is displayed. This is something of a mystery. Since the current process representation is always the target of

the **configure** command, there are no loops or searches involved with the call. The **configure** command need only retrieve a few values from the current process, load those values into the text boxes and menus of the configure dialog, and display the dialog. The problem may lie in the fact that DAVE creates a new configure dialog object each time the command is called, rather than reusing one object repeatedly.

Attach an algorithm to a process representation

Once the current *ProcessBean* has been configured, the user will most likely wish to attach a description of an I/O Automaton to the process representation. At the highest level of the object hierarchy, this is not a particularly interesting problem – the *DAVEDisplay* class creates a file dialog, extracts the file from that dialog, and then passes that file to the current *ProcessBean*.

In *ProcessBean*, things get much more interesting. The *ProcessBean* function **loadState** is crucial to the operation of DAVE. **loadState** is the front end to the function that loads an algorithm into the *ProcessBean*. This function, by itself, is not very impressive. It has much the same functionality as the other parser front ends in the *DAVEDisplay* class, with one notable difference. Every *ProcessBean* carries the text of its algorithm around internally. At this time, this feature is not fully exploited. However, future versions of the program will add a feature that allows the user to view the code for a given process. When this feature is implemented, this code storage functionality will become critical. This also opens the avenue for easier simulation. Since each process contains its IOA code, a simulator can run using the *ProcessBean*s alone, rather than being forced to refer to outside sources. This makes configuration storage as objects (rather than strings) a much more viable and useful option.

The next important function is the body of the algorithm loading functionality, **parsecode**, is also located in the *ProcessBean* class. Once the IOA code has been loaded into a *ProcessBean*, it will be necessary to extract certain information from it, namely state variables

and actions. The code itself is fairly straightforward, but the parser is rather primitive. Currently, DAVE expects lines to be delimited by a new line character. This is not in keeping with the standard established by the IOA manual. Using special characters to delimit the pieces of a line allows the function to easily extract the pieces necessary to assemble objects. For example, the name of a transition ends with a '(' character – DAVE uses this information to split a transition name into a name and a list of arguments. When extracting an action, **parsecode** must discern the action's name and its type. DAVE will later extract the arguments from the name in class *Transition*, but for now it needs only this information. There are different modes in the function – since every line is not flagged with its type (transition, state, etc.) it is necessary to keep track of which mode the function is currently in. When the program is in TRANSITION mode, and has extracted the necessary information from the current line, a new *Transition* is created and this object is added to the *ProcessBean*'s vector of transitions.

parsecode does not implement a fully-featured parser of the IOA language. **parsecode** can only extract *simple* automata. The IOA language also allows for composition of automata, a feature which **parsecode** does not deal with. **parsecode** also ignores the *type* statement in IOA, a very important aspect of the language. *type* statements can be very complex, and it seemed unnecessary to tackle that complexity, since all of the values in DAVE are specified by string. **parsecode** serves one very simple purpose – to extract the names of a simple automata's variables and transitions.

Connecting two actions

To compose two actions together, the user must use the **connect** tool. The first level of this tool is located in the *DAVEDisplay* class. The *DAVEDisplay* **connect** function contains little of the functionality that actually establishes a connection. The function merely sets a flag and changes the cursor to indicate to the user that she is now in connect mode. Most of the connect

functionality is contained with the *ProcessBean* class, and this functionality is accessed when a mouse event is dispatched that causes a connection to take place.

The connection process is continued in the **connect** function of *ProcessBean*. **connect** is passed a *ProcessBean* – the source of the connection, and a *Transition* that was selected by the user from the popup menu generated when the Connect tool was used. Using this *Transition* as a guide, **connect** iterates through the *Transitions* in the source *ProcessBean* until it finds a connection to make. The matching *Transition* must meet three criterion:

1.) The proper name – that is, the name of the *Transition* with the arguments removed – must be the same as that of the *Transition* returned by **getTransAt**.

2.) The source *Transition* must be of type *output*, and the destination *Transition* must be of type *input*.

3.) The *Composition* of these two *Transitions* must not already exist.

If these criteria are met, then a connection can be established. The last responsibility of **connect** is to assure that a *Connection* exists between the two processes. Between any two *ProcessBean* objects, there may exist at most two *Connections*. For any two *ProcessBeans* P1 and P2, a *Connection* can go from P1 to P2, and a different *Connection* from P2 to P1. If the *Connection* required to hold the new *Composition* does not exists, it is the responsibility of **connect** to create one. However, the *Connections* must be of two different types. For example, if the *Connection* from P1 to P2 and from P2 to P1 were the same color, it would be difficult to discern which *Compositions* belonged to which *Connections*. Thus, if a *Connection* from P1 to P2 does not yet exist, **connect** must make sure that a *Connection* from P2 to P1 does not exist. If it does, then the new *Connection* must be made of a different type, so it will display in a different

color than the first *Connection*. Once this *Connection* has been made, the new *Composition* can be added to the *Connection* and **connect** has completed its task.

Handling mouse events in the client area

As described in Section 3, much of the functionality of DAVE is accessed by simple mouse commands in the client area of the program. These mouse events are handled in the *DAVEDisplay* object, in the **processMouseEvent** function. This function is responsible for dispatching most of the commands for such essential use requests as moving processes, selecting the current process, and connecting processes. The first event handled is the popup trigger. This is the mouse event, defined by the system, that is supposed to bring up a popup menu (for example, clicking the right mouse button on a Windows system). In DAVE, this popup menu only contains two commands – configure and connect.

The next event handled is a single left-button click. This command can have several interpretations. If the user clicks on a hotspot, the event must be dispatched to **checkhotspot**, and the correct hotspot displayed. Otherwise, the user may be selecting the current process, in which case the event much be dispatched to **checkcurrent**. If neither of these is the case, the click may simply be an erroneous mouse click in the client area, in which case DAVE shouldn't do anything. All of the code to dispatch these events is straightforward, as the functionality associated with these user actions is encapsulated in other functions.

processMouseEvent also handles dragging processes in the client area. This is handled in the MOUSE_PRESSED and MOUSE_RELEASED sections of the function. However, this function must take into consideration that a mouse release may also indicate a button press. This is something it must check for – whether the mouse was dragged, or merely clicked. Thus, in MOUSE_RELEASED, some of the functionality already in place in MOUSE_CLICKED is

repeated. If the mouse was, in fact, dragged then the position of the current process must updated to the new mouse position. In this version of DAVE, the upper left corner of the process is moved to the current position of the mouse. This problem will be fixed in later versions to make the behavior more in keeping with standard Windows dragging behavior, where the part of the object that was "grabbed" by the mouse is moved to the mouse's coordinates. Any mouse event that is not handled by **processMouseEvent** is passed up to the superclass of *DAVEDisplay*.

The **checkcurrent** function handles the single mouse click that selects the current process. The x and y coordinates being used by this function are passed in from the mouse event handler. **checkcurrent** then scrolls through the vector of processes being maintained by *DAVEDisplay* until one is found that contains the x and y coordinates generated by the mouse click. This process is then set as current, and the former current process is un-set as current.

This is accomplished with the *ProcessBean* functions **make_current** and **unmake_current**, which merely set a boolean flag. These functions, however, are by no means trivial. When the current flag is set, the *ProcessBean* has many behaviors that it doesn't have otherwise. It has all the features of the current process, as described in Section 3. It is also displayed differently than other processes, as we will see later.

The functions **closeto** and **hitTest** of the *ProcessBean* class are also used for processing mouse events. When the *DAVEDisplay* object receives a mouse event, it has only the x and y coordinates. It must use these to figure out which process, if any, has actually been clicked on. When it wishes to do this, it must loop through the list of processes and call **hitTest** on each one. This is a boolean function that will return true if the mouse click is within the bounds of the *ProcessBean*. While this may seem crude, it is currently the most practical way to handle mouse clicks in an event driven system.

The function **hotspot**, quite simply, checks whether a given set of coordinates are within the bounds of a hotspot on some process or the hotspot of some variable within a process. If so, it calls the appropriate function to display the contents of that hotpot. Notice that both *ProcessBean* and *Variable* implement the **inSpot** function. This is not required – that is, they have no common superclass. Rather, it was simply a convenient naming function that makes the code appear more coherent.

The function **DisplayState** is responsible for creating the state viewer that pops up when one of the "hotspots" is clicked (the "hotspots" were generated when the list of a process's state variables exceeded the bounds of the process circle). To display the state, DAVE uses a *TextArea* object. This object takes in a single string and displays it in a window. Figure 8 shows a **DisplayState** dialog containing the current state of a Channel process. Since *TextArea* must have a single string to work with, notice that **DisplayState** takes the entire state of the process, formats it with the appropriate labels, and then concatenates the entire state together into one string. Again, **GridBagLayout** is used to manage the layout of our dialog. This is due to the fact that DAVE must put a very large text window and a fairly small button in the same dialog.

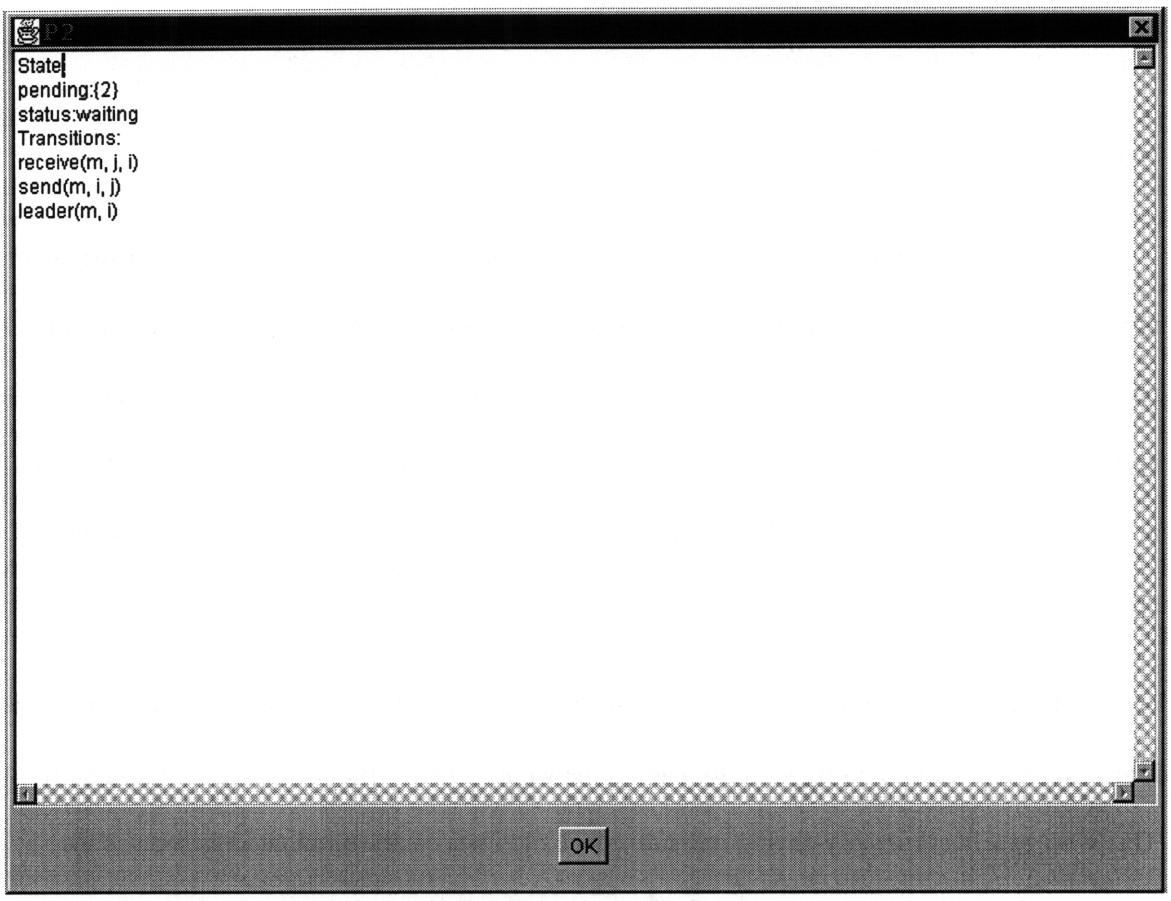

Figure 8: The "View State" dialog.

The **DisplayVariable** function serves much the same purpose as the **DisplayState** function, only on individual variables instead of entire processes. Recall that hotspots are also put on variables that had either oddly typed values (i.e. images), or values that were simply too long to show inside the process circle. In either case, a specialized viewer needs to appear when the hotspot is clicked. In the current implementation, only the latter difficulty is handled – particularly long values. Future versions of DAVE will handle a broader set of data types, and the handlers for those types will have to pass through **DisplayVariable**. In almost every respect, **DisplayVariable**'s functionality is identical to that of **DisplayState**.

Save and Load Configuration

The process of saving and loading configurations – the visual layout of the client area of DAVE – begins in the *DAVEDisplay* class. **load_config** is the function responsible for parsing the first type of save file – the *.cfg* file that is written out by unparsing every object in the system. Only DAVE itself will be responsible for writing out *.cfg* files, which means that as long as the process of saving and loading configurations is consistent within DAVE, there should never be a problem. A *.cfg* file is not something that the user would wish to write out by hand.

load_config begins by opening the file and loading the first line. **load_config** processes lines as they are read in, as opposed to loading the whole text and then working on it as a body. This is an excellent memory-saving measure, since old lines can be eliminated as soon as the corresponding process has been created. When **load_config** has a line from the file, it extracts the first token from the line, called a *type* identifier. Depending on this *type*, the line is processed in very different ways. If the *type* is "PROCESS", **load_config** must extract the necessary information from the line to create a new process. This includes the x and y coordinates, the width and height, the color, font, and label of the process, and the algorithm file associated with the process. Once it has the information, DAVE creates the process very much as if a user had called "new process" and then configured the process. **load_config** is also responsible for loading the process's algorithm file.

If the *type* is "CONNECTION", on the other hand, **load_config** has only the name of the connection to work with. However, as demonstrated in Section 3, this is enough. DAVE extracts the name of the action being composed and the labels of the involved process from the line. **load_config** then loops through *DAVEDisplay*'s internal list of processes until it finds both the source and destination processes, and then loops through the destination process's actions until it

finds the correct action. With this information in hand, **load_config** can call the **connect** function at the source process and create the connection.

One thing we must keep in mind about *.cfg* files is that all of the process statements must precede the composition statements. While this may seem like an inconvenience, it is really of no concern to the user. Since only DAVE will be writing these *.cfg* files, we are free to impose whatever restrictions on the format we require.

save_config is **load_config**'s counterpart, and is responsible for writing *.cfg* files. In many ways, **save_config** behaves as the reverse of **load_config**. In **save_config**, DAVE cycles through all of the processes in *DAVEDisplay*'s internal vector of processes and creates a string containing the unparses of the process's x and y coordinates, width, height, color, font, and algorithm. This string is then written to a streaming output. Once all of the process information has been written, the program loops through the processes and extracts the composition information. This information is also written to the output stream. Once the function is complete, the *cfg* information is flushed from the output stream and the function returns.

Load an execution

Once the user has the configuration she wishes to study, she will wish to load an *execution*. As discussed in Section 3, an execution contains the states of an I/O Automaton and the transitions between states. The process of loading an execution begins in *DAVEDisplay* with a function called **loadExecution**. **loadExecution** reads in an *.ext* file and prepares the execution for display. The file I/O for this function is very much like the I/O for **load_config**, with one notable exception. In **loadExecution**, the entire file is read and stores in memory. This is because the user will wish to move forward and backwards freely through the execution. However, this is certainly not the optimal way to store an execution. DAVE's performance woul

undoubtedly improve if the states listed in an execution were stored in the *ProcessBean* objects themselves, rather than keeping the entire text of a *.ext* file sitting in memory.

Section II stated that all executions begin with an initial state. This is the responsibility of **init()**. **init()** is called by **loadExecution** once the text of an execution has been loaded. **init()** contains the second of DAVE's three parsers. Section 3 outlined the format of the *.ext* file, and **init** parses this format rather handily. **init** makes sure that the first line in the buffer is a "STATE:" statement, and then parses lines until it reaches a "TRANSITION:" statement. **init** takes a line from the buffer containing the name of the process, and then loops through the file until it reaches a new process declaration. These lines contain variable names and values, which are easily extracted because they are delimited by special characters. Once **init** has this information in hand, it calls **SetVarValue** with the process name, variable name, and value to set the new value.

Play an execution

The **faster** and **slower** functions of *DAVEDisplay* control the thread that controls playback. Suffice to say that these functions are very simple – they simply increment or decrement the floating point threshold at which the player thread steps an execution. This functionality was added more as a debugging tool, a stop-gap measure until a more stable system could be implemented.

The last class in DAVE.java is the *Player* class, which is responsible for playing back an execution by calling the appropriate functions at timed intervals. This class implements the interface *Runnable*. All this means is that we have implemented the virtual function **run**. This function is exactly like the function **main** – it dictates a sequence of instructions to be run in the new thread. Because DAVE uses an event driven model, there is no constant loop processing the

user inputs. While this provides a great deal of freedom, it also means that there is no way to do timed playback from the body of our program.

Playing the execution (that is, stepping at fixed time intervals, much like a CD or cassette playback) requires the creation of a new thread of execution. This thread has one responsibility – to step the execution at certain intervals after the user presses the play button. To create a thread in Java, the thread requires *Runnable* object. This is where *Player* comes in.

The most important function in *Player* is the **run** function. All that **run** does is increment a counter at every computational cycle. When this counter reaches a certain threshold, *Player* calls the step function of the *DAVEDisplay* object that was passed to it. Of course, this functionality must be turned on and off, which is why control of the *Player* object has been given to the *DAVE* class, which controls, among other things, the **play** button. This is not the most efficient system for playing back an execution, since the use of a counter uses considerable processor time. A more efficient system would be to put the *Player* thread to sleep when it isn't in use. This functionality should be easy to add to the DAVE system.

Whenever the **play** button is pushed in the DAVE window, this will cause an event to fire which will set a boolean flag within DAVE. As the *Player* thread is processing, it will check at every cycle whether this flag has been set. If the flag is set (that is, the console should currently be playing), the *Player* will begin incrementing, and call **next** on the DAVE class whenever it reaches its threshold value. If the user presses **pause**, the flag inside of DAVE is un-set, and *Player* no longer increments or calls **next** until the flag is set again.

The other three functions in *Player* are a creation function, an **increment** function, and a **decrement** function. The creation function is passed a *DAVEDisplay* object as an argument. This will give the *Player* access to the **next** function it requires. The other two functions are used to increment or decrement the threshold value. This has the effect of making the playback go

faster or slower. This thresholding system should be replaced with an actual clock, to remove the system dependency of the playback speed.

The behavior of the *Player* thread depends on a single boolean variable that is controlled by the *DAVEDisplay* functions **play**(), **pause**(), and **isplaying**(). **play** and **pause** simply set this boolean value, which is then accessed by the player thread through the **isplaying** function. It could be argued that the **isplaying** function is not required, and that it would be easier to simply make *play* a public variable. However, this would violate DAVE's strict adherence to the precepts of object-oriented programming. This phenomenon of hiding simple variables behind function interfaces will crop up over and over in the DAVE code. The purpose behind doing this is to simplify the use of DAVE classes for future programmers.

Much of the functionality of the *DAVEDisplay* function **init** is also a part of the basic function of playback, the *DAVEDisplay* **next. next** is the function responsible for "stepping through" the execution. When **next** is called, the pending transition is highlighted in the DAVE window and its arguments are set. Then the next state is read and loaded into the processes. First, **next** resets the system, using the **reset** function. This means that the connection that was last highlighted is set back to its inactive mode. The arguments of all actions are reset. Then **next** reads the next transition line and calls **SetTransValue** with the source and destination process names, the transition name, and the argument values. **next** then moves forward in the execution and executes the functionality of **init**, reading in the new state. After this function is called, the entire canvas is repainted. This is necessary because an entirely new set of variables will be highlighted, and a new connection has become active. This information must be relayed to the user through color changes on the canvas. Ideally, DAVE should only repaint the portions of the canvas that are changed. However, this optimization has not yet been added to DAVE.

The *ProcessBean* **SetTransValue** and **SetVarValue**, are the the counterparts of the same functions in *DAVEDisplay*. A *Variable* value can not be changed from the *DAVEDisplay* object. This is because *DAVEDisplay* has little access to the internals of the *ProcessBean* object. Thus, *DAVEDisplay* could merely identify the *ProcessBean* involved, and then relegate the duty of changing the variable values to that *ProcessBean*. The *ProcessBean* then uses these functions to change the appropriate value. Note, however, that the same separation takes place in these functions. *ProcessBean* can merely identify the *Variable* or *Transition* involved in the operation by simple name matching. Once that object is identified, the responsibility passes to the object to make the appropriate changes.

next's sister function is **previous**. This function is much easier to implement, since **next** is called in the body of the function. Rather than trying to step backwards in the execution text, **previous** merely moves the execution pointer back to the previous transition and then calls **next**. This has the effect of "backing up" the execution to its previous step. It now becomes apparent why the entire state of every process is declared in the *.ext* file – it saves a great deal of effort when attempting to step backwards through the execution.

The next function that is relevant to playing an execution is **reset**, a member function of *DAVEDisplay*. The need for **reset** has already been described above, and its functionality is remarkably simple. **reset** prepares the client area of DAVE to display another step of an execution. This involves undoing the color changes – the highlighted variables and links – that were made when the last step of the execution was processed. Because of the tree structure shown in Figure 7, a great deal of *DAVEDisplay*'s functionality to can be relegated to lower components in the graph. This is done in **reset**. **reset** simply scrolls through *DAVEDisplay*'s list of processes and calls **reset** on each of these. The **reset** present in *ProcessBean* is not nearly this simple.

One place where this rigorous separation of levels was not maintained is in the **reset** function of *ProcessBean*. The purpose of **reset** is to prepare all of the graphical elements for the next step of the execution by un-setting the "modified" flags of all elements. Since changed elements are highlighted in red when viewing an execution, it is necessary to reset the old elements before proceeding. The **reset** function of *ProcessBean* is responsible for doing this. This function iterates through **myState**, **myTransitions**, and **myConnections** and resets each element by un-setting the "modified" flag of each element. When this is done, execution playback can proceed.

Draw the DAVEDisplay

The **paint** function for *DAVEDisplay* is very simple, as each object in the program (e.g. process, variable) is responsible for painting itself. Since this is the case, DAVE need only paint all of the processes in its process vector. The call to **paint** will then branch out as seen in Figure 7, as each object calls the paint function of its child objects. This means that the **paint** command will then branch out to the *ProcessBeans* associated with the *DAVEDisplay* object.

The painting function of *ProcessBean* is the **draw** function. Almost all of the graphical horsepower of DAVE is derived from this function. The top-level *DAVEDisplay* object has no real control over the appearance of the client area – the **draw** functions of the individual *ProcessBeans* dictate the layout of the workspace. The first responsibility of **draw** is to draw the *ProcessBean* itself. This is accomplished by using AWT routines to draw the label of the process and its label. Recall, however, that if the *ProcessBean* is the current process, flags are set such that the outline of the oval will be drawn in red, indicating the *ProcessBean*'s special status to the user.

Once the *ProcessBean* itself is drawn, it must also draw all of its child components. Based on the current position of the *ProcessBean*, the *Variable* objects in **myState** are configured accordingly. This involved setting the x and y coordinates of the variable on the screen and deciding whether the variable is long enough require a value popup tab. Once the *Variable*s are configured, they are responsible for drawing themselves. A similar hierarchy exists for the *Transition* objects in **myTransitions**. The notable exception to this format is the drawing of *Connection* objects. These objects need no configuration –that is, the *ProcessBean* is not responsible for calling a function that will set the *Connection*s position on the screen. The *ProcessBean*'s **draw** function instead calls the **paint** function of the *Connection*s the *ProcessBean* contains. The *Connection*s then set their own x and y screen coordinates, based on the screen coordinates of their parent *ProcessBean*s. This is possible because each *Connection* contains its parent *ProcessBean*s, and can access information about the location of the parents.

Comments on DAVEDisplay and ProcessBean

Throughout *DAVEDisplay* and *ProcessBean*, there are several recurring themes. The first of these is the benefit derived from using an event driven system. This system makes it easy to implement graphical components such as buttons and dialogs, since each component only requires creation and a handler function. The second benefit is derived from the hierarchical object system used in DAVE. Throughout *DAVEDisplay* and *ProcessBean*, functionality is relegated to lower, smaller modules, where the complexity is more easily managed.

Because of the hierarchical, object-oriented structure of DAVE, the highest level of the program is exceedingly simple. Surprisingly, the lowest levels of the tree – the *Variable*s, *Transition*s, and *Compositions* – are also very simple. Most of the functionality of DAVE is located in the *ProcessBean* and *DAVEDisplay* classes. These classes, while complex, are never

unmanageably so. The code for these classes is very long because these classes serve as the middlemen between the high-level execution of the program and the low-level details of the smaller objects – *Variable*s, *Compositions*, *Connections*, and *Transition*s.

Smaller Types: Variable

The first of these objects is the *Variable* type. It is easy to dismiss this class simply as a wrapper around a name and value. However, this is hardly the case. A significant amount of functionality relating to the *Variable* was moved to where it belongs – within the *Variable* object itself. A *Variable* represents a state variable and contains, most importantly, the name and value of a state variable.

Most of the functionality of the *Variable* class has already been explored in earlier sections of this paper. Much of the remainder of the functionality is, again, a set of interfaces to maintain our strict adherence to object-oriented principles. This section concentrates on the most interesting aspects of the *Variable* class.

The first function of interest is the function **hitTest**. This is the lowest rung on the ladder that began in *DAVEDisplay* – namely, finding out which *Variable* is actually being clicked on. Recall that, at each level of mouse event processing, DAVE strips away a layer. The progression follows from a mouse event, to a particular process, and now to the actual variable through a series of function calls. This chain starts at the mouse event handler for *DAVEDisplay*, which calls a function within the *ProcessBean* that was clicked, which in turn calls a function in the *Variable* that was clicked. The "click" is represented by a set of screen coordinates that is passed down through the levels of the program. This exemplifies the hierarchical character of DAVE that has been stressed throughout this document.

The next function, **configure**, plays a large role in the "hotspotting" of *Variables*. This function decides whether a *Variable* should be hotspotted or not, and sets the appropriate flags. Remember that this function was called every time the parent *ProcessBean* was painted. DAVE does this because the decision to hotspot a *Variable* may change between every repaint.

The last function of interest in class *Variable* is, of course, the **draw** function. This function is remarkably simple – it merely formats and prints the *Variable* name and value at the object's coordinates. This simplicity is achieved because much of the work was already done at the layer above. The *ProcessBean* is responsible for positioning the *Variable* because the *Variable* has no knowledge of its parent's structure. Thus, when the time comes to draw the *Variable* object, it is in the proper position.

Smaller Types: Transition

A close relative to the *Variable* class is the *Transition* class – the two are very similar in structure. The only notable difference is the **setArgs** function of class *Transition*. This function is used when the user is viewing an execution. When a step is taken (that is, a composed action is executed or an internal action is read from the execution), the user may want the component actions of that transition to show the arguments to the action. That is why the *Transition* class uses **setArgs**. This will (temporarily) set the text of the action such that the variable values are present in place of the argument names. To undo this effect when the execution proceeds to the next step, DAVE uses **unsetArgs**. This function reverts the *Transition* to its original name, with argument labels in place.

Smaller Types: Composition and Connection

The last two classes in DAVE, *Composition* and *Connection*, are very closely related. *Composition* contains two *Transitions* and a name. These are the source and destination actions of the composition, and the complete name (e.g. send(m)-P1-P2) of the composition. There is really very little else involved in the *Composition*, since the responsibility of drawing the *Composition* falls to the *Connection* object.

Because *Composition* does so little, *Connection* is a somewhat complex class. The *Connection* represents a conduit between two processes through which the *Compositions*, like wires, flow. A *Connection* contains two *ProcessBeans* – a source and a destination. These are set when the *Connection* is created. A *Connection* also contains a vector of compositons – the "wires" running between the processes. *Connection* contains several functions necessary for maintaining this vector of *Compositions*, but these functions are straightforward and easy to understand.

Connections have a distinct *type*. This is explained earlier in the section about **connect**. The type is required so that *Connections* that run between the same two processes can be distinguished from one another in the client area by being painted in different colors. To achieve this functionality, *Connection* contains a command **setType**, which gives the *ProcessBeans* responsible for creating the *Connections* the ability to set their graphical characteristics. To aid in establishing whether a *Connection* does exist between two *ProcessBeans* during the **connection** process, *Connections* also implement a function to test for the existence of a *Connection* between any two *ProcessBeans*.

Connections are the most graphically complex objects in the client area of DAVE, since they must connect between points and place a label at the midpoint of the line. This complexity can be seen in the **draw** function. The length appears daunting, but the function itself is quite

simple – most of the arithmetic gymnastics are necessary to find the midpoint of the line and display the labels appropriately.

The study of the structure of the DAVE code has revealed several interesting aspects of the program that bear repeating. A precept harped upon since the beginning is a rigorous devotion to the principles of object-oriented programming. This system made the entire development process orders of magnitude easier, and allows the smooth introduction of any modifications. DAVE also adheres to a strict hierarchical structure. This hierarchy is imposed, for the most part, by the object-oriented nature of the programming.

Section V: Sample Execution

DAVE is a very visual program, and as such has no real numerical results. Instead, we will walk through a typical user interaction with the system, demonstrating along the way the powerful visualization functionality of DAVE.

Appendix 1 contains the code that we will be using in this example. The two algorithms, *process.alg* and *channel.alg*, represent a process involved in an LCR leader election algorithm (see page 28 of Lynch) and a typical, one-way FIFO channel, respectively. The execution, *LCR.ext*, represents the execution of a four-process system running the LCR leader election algorithm.

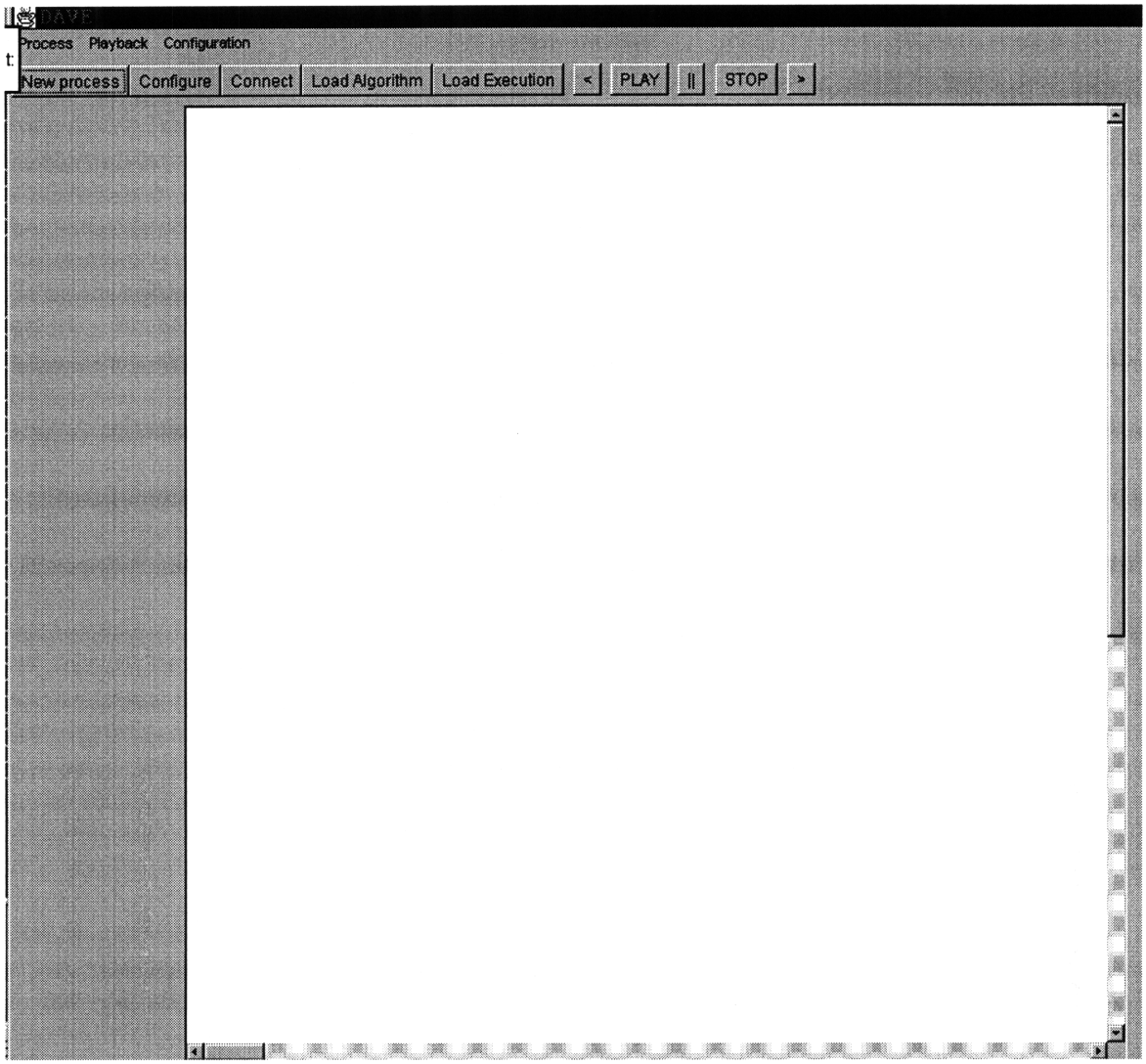

Figure 9: A fresh DAVE Window.

The user begins with an empty DAVE window, as shown in Figure 9. To begin the construction process, the user clicks the "New Process" button. This command creates a new process in the center of the screen. This process will be the new current process. To configure this process, the user need only click the "Configure" button, or select this command from the popup menu. Figure 10 shows the user's progress at this point, with the Configuration dialog box open. In this figure, the user has already entered the data relevant to this process.

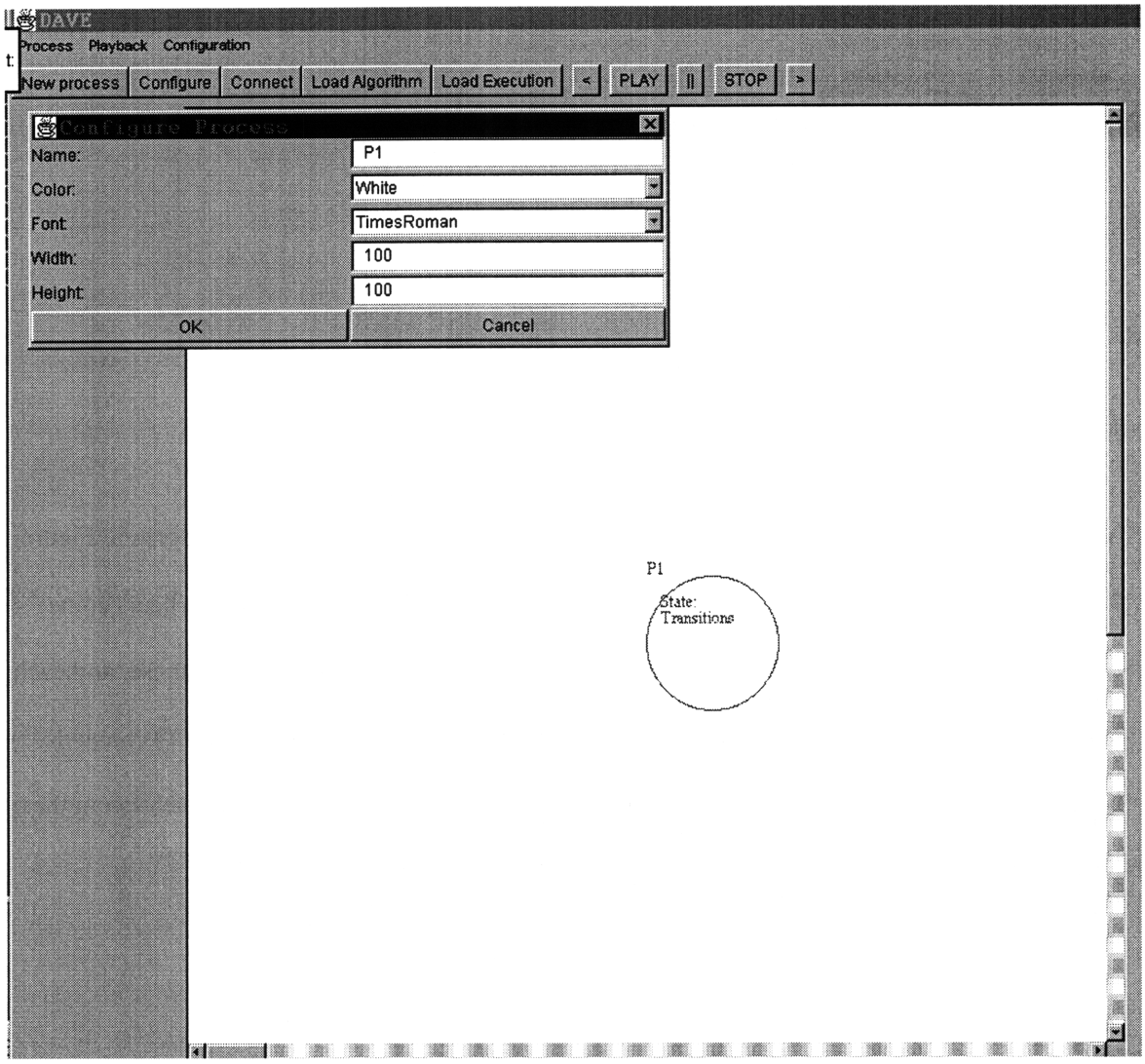

Figure 10: The user configures the new process.

Once the user has properly configured the process, she is ready to attach an algorithm to the process. The user selects the "Load Algorithm" command and selects a *.alg* file from the list of available files. In this case, the user wishes to create a *Process* process, and thus selects *Process.alg*. Figure 11 shows the process with the new state loaded. Notice that none of the variables have values associated with them.

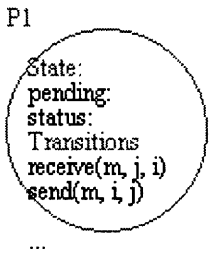

Figure 11: The process, now attached to the Process automaton.

The user repeats this process until all of the desired processes are on the screen. Figure 12 shows the current configuration of the IOA. This diagram is complicated, and bears some explanation. The yellow processes are the *Process* automata. There are three of these, and each has been loaded with the *Process.alg* code. The green processes represent the FIFO channels. Notice that between every pair of processes, there is a channel process. Communication in the LCR leader election algorithm only moves one way around the ring.

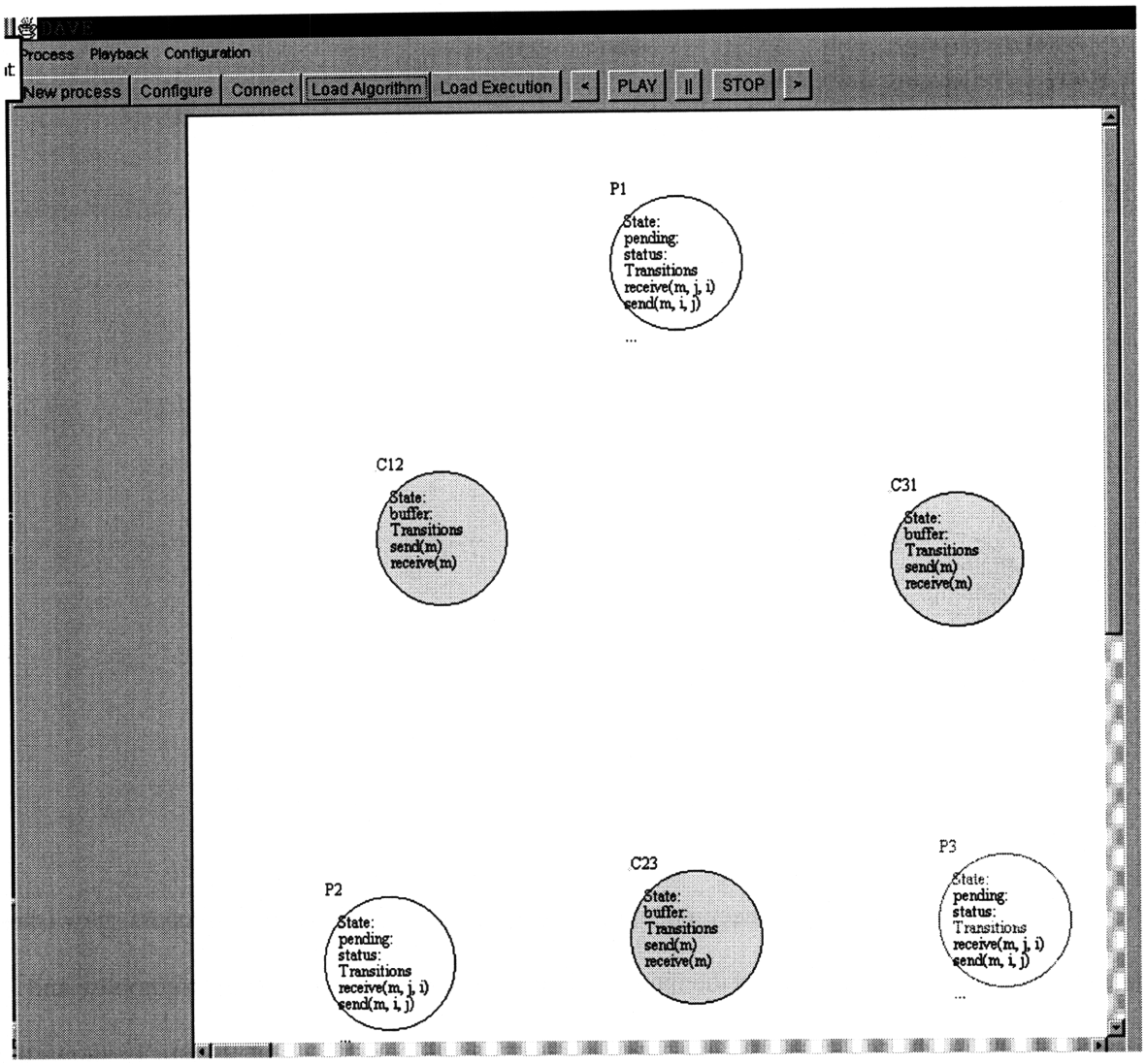

Figure 12: The processes in place.

Now that the processes are in place and have algorithms loaded, the user will begin composing functions together to create the finished IOA. The user begins by selecting a current process to connect. This process will be the *source* process for the composition. The user then clicks the Connect button or selects that command from a popup menu.

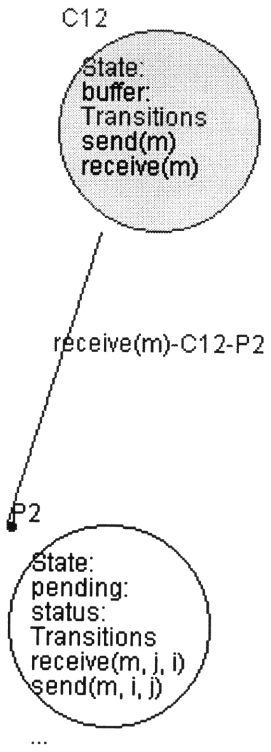

Figure 13: A new connection

Once the user is in Connect mode, she moves the mouse pointer to the process she wishes to connect with. She then centers the mouse pointer over the action she wishes to compose and clicks the mouse button. At this point, a new connection will be established. Figure 13 shows the new connection. The destination process is the *Channel* C12, and the source is *Process* P1. The action that has been composed is the *send* action, which takes one argument, *m*. All of this information can be extracted from the label on the *Connection* that has just been formed. Note also the direction of the connection between the two processes – this large dot indicates the destination of the composition.

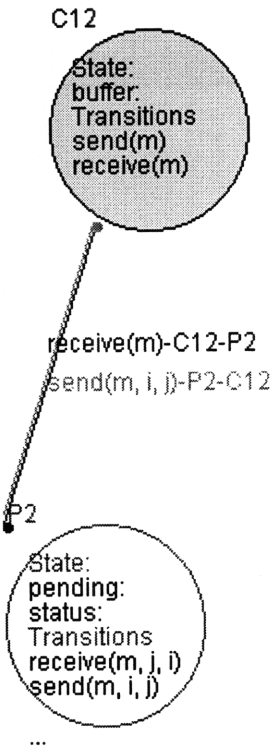

Figure 14: Two connections. Note the color difference between the two.

Say the user decides to connect the *receive* action of C1 back to the *receive* action of P1. This is not technically correct for this example, but it does point out an important feature of DAVE. Figure 14 shows the new diagram with both connections in place. The original connection is still blue. However, the new connection and its label are drawn in green. This convention allows the user to distinguish between the compositions of two different connections, even when the connections are very close to one another.

But, to return to the LCR example, the user has completed making all of the connections she wishes to. Figure 15 shows the completed diagram. The user had moved several of the processes to allow for easier viewing of the diagram. After this amount of work, the user would most likely want to save her work. At the end of Appendix 1 is the text of the *.cfg* generated by executing the Save Configuration command on this IOA.

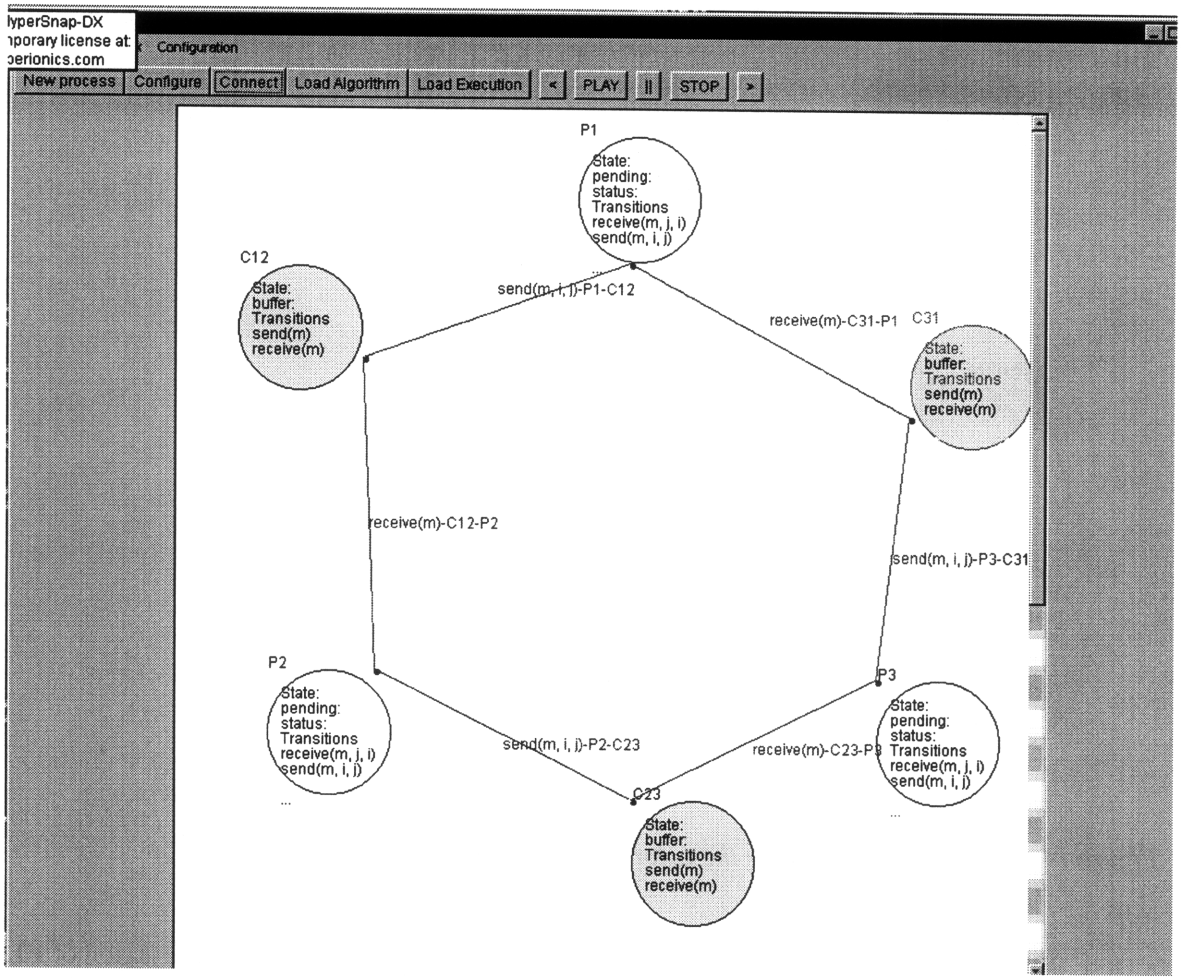

Figure 15: The completed I/O Automaton representation

At this point, the user may wish to send this newly constructed IOA to a simulator. That simulator would return an execution of the IOA. The user will now take advantage of the second half of DAVE's functionality – the ability to view executions.

If the user has started a new session, she will load the configuration she saved in the first part of our example. The next step is to use the Load Execution command. This generates a file dialog similar to the Load Algorithm and Load Configuration commands, from which the user selects the appropriate *.ext* file. The execution will be loaded and initialized, as discussed in Section 4. Figure 16 shows the state of the execution after the Load Execution command has completed. Every variable in every process now has a value attached to it. For example, in

Figure 15, variable *pending* in process P2 is blank. In Figure 16, that variable now has the value **{2}** following its name.

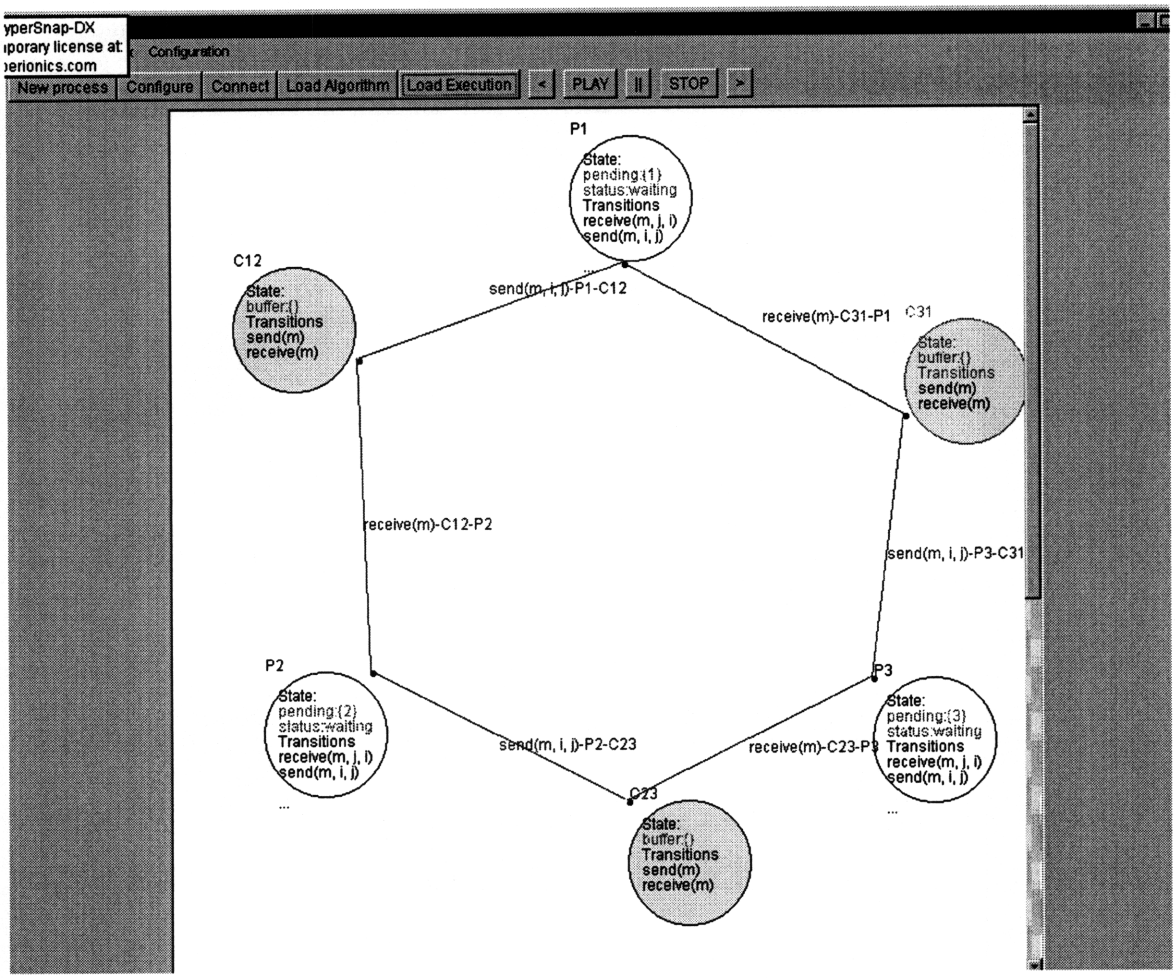

Figure 16: The initial state of the automaton.

At this point, the user has several options. She may begin stepping through the execution by using the **Step Forward** command, or she may simply press **Play**. These commands are located on the right end of the toolbar, and resemble a cassette deck console. In this case, assume that the user has pressed Play. The execution will then proceed at approximately one second intervals. Figures 17 and 18 show two subsequent steps of the execution. These steps are highlighted in the execution listing in Appendix 1. Figure 17 shows process P2 receiving a message from Channel C12. The arguments to the composition and its two component action have been filled in. In Figure 18, Process 3 is sending a message to P1, via channel C31. The

buffer of channel C31 now contains the UID of Process P3. Again, the arguments of the last composition executed and its component transitions are filled in.

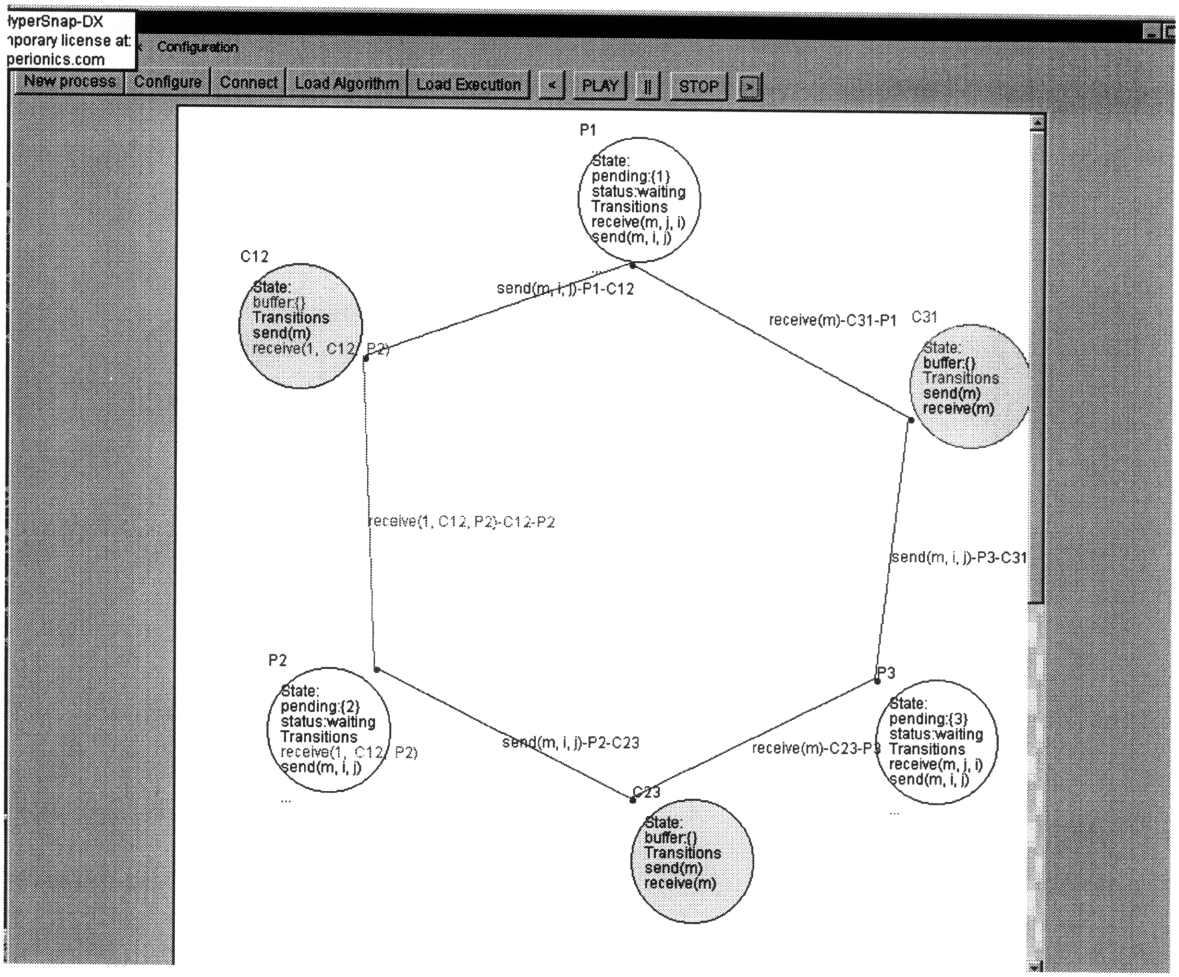

Figure 17: Two steps into the execution.

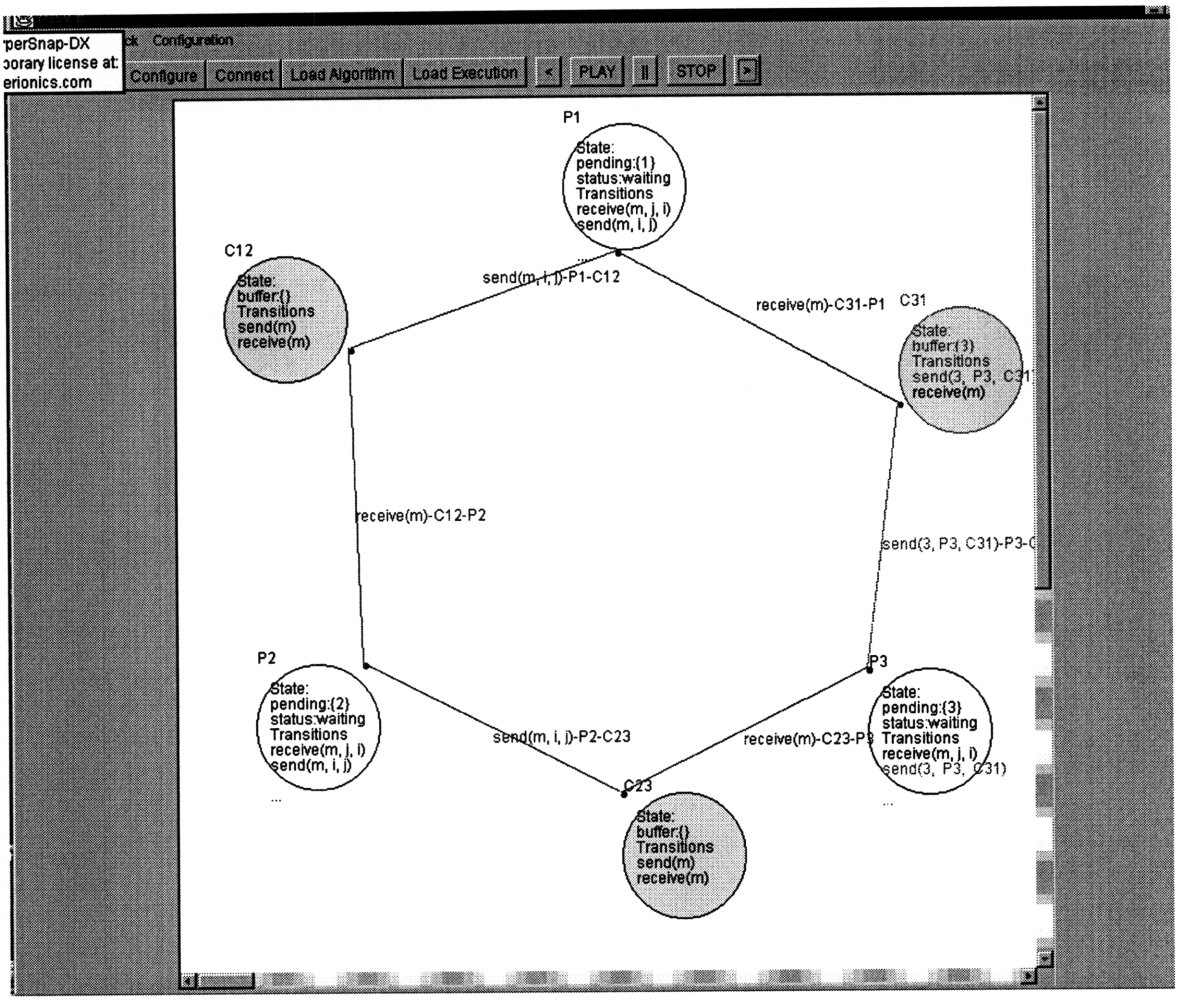

Figure 18: Three steps into the execution.

The execution playback controls give the user the ability to freely move back and forth in the execution. Eventually, the user may reach the end of the execution. Figure 19 shows the execution in its final state. This example highlights most of the functionality of DAVE. Of course, there are several possible paths of execution within the program, and not all of them can be explored here.

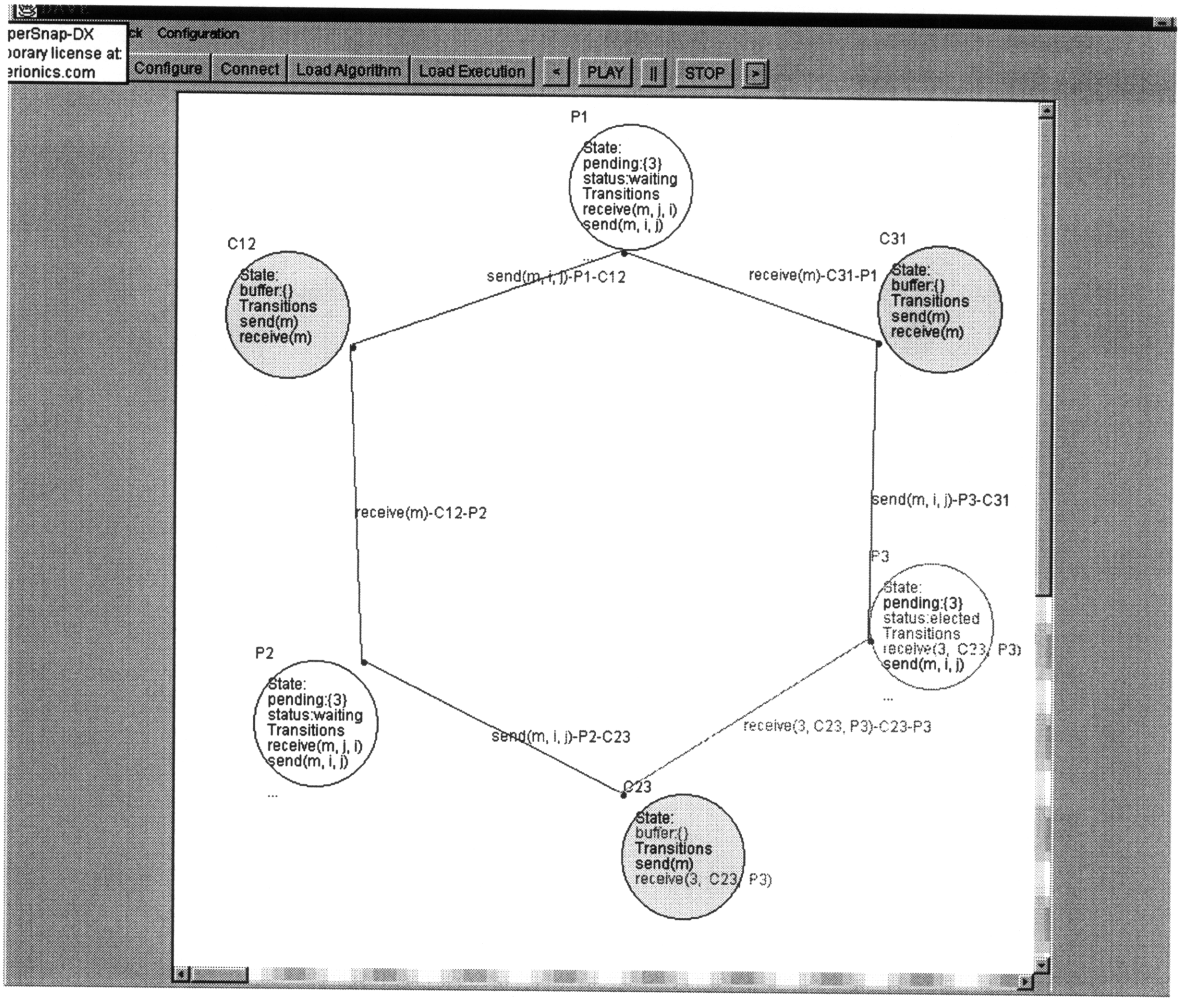

Figure 19: The final state of the execution. Notice that P3 has been elected leader.

We can derive some numerical data from the DAVE program. Once of gravest concerns about Java programs in general is that they generally perform slowly. Table 1 shows the average execution time for some simple DAVE commands. The system used to take these measurements was a Pentium 166 with 80 MB of RAM running the Windows 95 operating system. The Java interpreter used was *java.exe*, part of the Java Development kit released by Sun Microsystems.

Operation	Average execution time
Create a new process	0.9 seconds
Connect two processes	1.2 seconds
Attach an automaton to a process	3.1 seconds
Load a 9 process configuration	10.8 seconds
Load an execution	5.2 seconds
Step forward in the execution	0.7 seconds

Table 1: Average execution times for some common DAVE tools.

These times are exceedingly slow, a problem that was not noticed until DAVE was nearly finished. Performance, initially not a concern with DAVE, has become its greatest problem. There are a number of reasons for this poor performance. The code has not been optimized – for example, there are several time-consuming loops in DAVE that could be circumvented. There may also be performance issues in the use of Java Beans – experience has shown that Beans have a tendency to be slow. The speed issue must be addressed before DAVE can be widely used.

Section VI: Conclusions

The example in Section 5 is intended to demonstrate that DAVE meets all of the design criterion and purposes set forth in Section 1 and Section 3. DAVE can be used to construct an arbitrarily complex IOA and view executions of that IOA. It is also a stated goal of this project that the underlying code conforms very rigidly to several design decision specified in Section 4.

Can DAVE be used as an effective learning tool? The user interface of DAVE is designed to be intuitive and familiar. A user that is somewhat familiar with the Windows environment should be able to understand the tools that have been provided with this software package. The client area of DAVE is also designed be familiar to any student of distributed algorithms, particularly students using Professor Nancy Lynch's Distributed Algorithms text. The graphical representations of processes and compositions are pulled straight from the pages of that text.

Given that this document has established the usability and familiarity of the DAVE interface, the only outstanding issue is that of performance. Does DAVE meet our a baseline performance goal? Based on the example in Section V, it seems that DAVE meets those goals quite handily. DAVE is not a performance-driven application – that is, it is not written with a great deal of computational shortcuts, obsessive optimization, or pure number-crunching horsepower. These goals are better left to the realm of simulators. The purpose of DAVE is to provide the user with the tools needed to construct any IOA she wishes to view, and free the user from any hindrance that would prevent her from achieving this goal.

Thus we arrive again at the issue of freedom. It is apparent throughout the design and implementation of DAVE that the program closes very few doors. The languages for algorithms and executions are well specified. However, these need not be the only languages supported by DAVE. DAVE was designed for the developer as well as the user. The structure of the code is designed to be a modular as possible, allowing future developers to add, remove, or change functionality as they see fit, and as the study of I/O Automata progresses.

DAVE is designed to provide the user with as much freedom as possible. Notice that there are very few limitations imposed by DAVE. Yes, processes must be represented by ovals or circles. The purpose of this decision, however, is not to limit the ability of the user to achieve the desired graphical representation of her IOA, but rather to enable the user by providing a common graphical language for I/O Automata. But even this standard can be changed with a small effort

by a single developer – significant change in the appearance of DAVE can be achieved in as little as half an hour.

DAVE is designed as the front end to an IOA simulator. Considering that such a simulator does not exist, proving that this goal has been achieved is something of a challenge. Nonetheless, refer back to the example of Section 5. Once the user has created an IOA, she may save that IOA to a file. This file, a *.cfg* file, can easily be read by a user or programmer. Without any knowledge of the DAVE system at all, the developers of a simulator could read this file as input, and provide DAVE with an output. Of course, this output may not be supported by the current implementation of DAVE. But again, the structure of DAVE is such that implementing a new parser for execution files is almost a matter of just dropping it into the current DAVE code.

As trite as it may sound, the results seen in Section 5 speak for themselves. The power of DAVE is evident from the example – the fact that the user can compose arbitrary automata into an IOA, and then run an execution through that system. It was postulated in Section 1 that one of the most difficult aspects of distributed algorithms is visualizing the execution of an algorithm. It is hoped that that the goal of simplifying this often challenging task has been met.

DAVE also has several notable problems, the worst of which is the speed issue. DAVE is unforgivably slow, and needs to be optimized. The DAVE IOA parser also needs to be fleshed out into a fully functional parser that can handle any IOA program. There are also a number of smaller problems – for example, the problem with resizing – that need to be fixed before DAVE can be used in a classroom or research environment.

DAVE can go in a myriad of different directions with future work. The most pressing issue is to develop a simulator to work under DAVE. After we have this simulator, though, we can begin expanding the DAVE program to serve the user better. DAVE should be expanded to be more compatible, and contain more tools to work with, the IOA language. The existing parser for IOA that is contained in DAVE is very crude. A library of pre-configured processes could be useful to the user. In addition, future developers could implement many features that were

beyond the scope of this project. This includes the ability to zoom the view in and out, as well as a series of "power tools" – tools that are designed to simplify the process of laying out automata. These tools could include a utility for assembling automata into common configurations: rings, grids, super-connected meshes, etc.

Because of the structure of DAVE, making these expansions is convenient and modular. Because of the extensibility and the platform independence of DAVE, the program will hopefully be in use for a long time. The field of distributed algorithms will always be challenging, and DAVE can be used by students and researchers alike to overcome that challenge.

Bibliography:

Garland, Stephen J., Nancy A. Lynch, and Mandana Vaziri.
IOA: A Language for Specifying, Programming, and Validating Distributed Systems.
MIT Laboratory for Computer Science. Cambridge, 1997.

Lynch, Nancy. Distributed Algorithms. Morgan Kaufman. San Francisco, 1996.

Appendix 1:

The process algorithm:

```
automaton Process(I:type, i:I)
      assumes RingIndex(I, String)
      type Status = enumeration of waiting, elected, announced
      signature
            input receive(m: String, const left(i), const i)
            output send(m: String, const i, const right(i)),
                  leader(m:String, const i)
      states
            pending: Mset[String] := {name(i)},
            status: Status := waiting
      transitions
            input receive(m, j, i)
            output send(m, i, j)
            output leader(m, i)
```

The channel algorithm:

```
automaton Channel
      uses SubSequence(M)
      signature
            input send(m:M)
            output receive(m:M)
      states
            buffer: Seq[M] := {}
      transitions
            input send(m)
                  eff buffer := buffer + m
            output receive(m)
                  pre buffer != {} && m = head(buffer)
                  eff buffer := tail(buffer)
```

The execution from Section 5:

```
STATE:
      P1:
            pending:={1}
            status:=waiting
      P2:
            pending:={2}
            status:=waiting
      P3:
            pending:={3}
            status:=waiting
      C12:
            buffer:={}
      C21:
            buffer:={}
      C23:
            buffer:={}
```

```
      C32:
            buffer:={}
      C31:
            buffer:={}
      C13:
            buffer:={}
TRANSITION:
      send(1, P1, C12)-P1-C12
STATE:
      P1:
            pending:={1}
            status:=waiting
      P2:
            pending:={2}
            status:=waiting
      P3:
            pending:={3}
            status:=waiting
      C12:
            buffer:={1}
      C21:
            buffer:={}
      C23:
            buffer:={}
      C32:
            buffer:={}
      C31:
            buffer:={}
      C13:
            buffer:={}
TRANSITION:
      receive(1, C12, P2)-C12-P2
STATE:
      P1:
            pending:={1}
            status:=waiting
      P2:
            pending:={2}
            status:=waiting
      P3:
            pending:={3}
            status:=waiting
      C12:
            buffer:={}
      C21:
            buffer:={}
      C23:
            buffer:={}
      C32:
            buffer:={}
      C31:
            buffer:={}
      C13:
            buffer:={}
TRANSITION:
      send(3, P3, C31)-P3-C31
STATE:
      P1:
```

```
            pending:={1}
            status:=waiting
      P2:
            pending:={2}
            status:=waiting
      P3:
            pending:={3}
            status:=waiting
      C12:
            buffer:={}
      C21:
            buffer:={}
      C23:
            buffer:={}
      C32:
            buffer:={}
      C31:
            buffer:={3}
      C13:
            buffer:={}
TRANSITION:
      receive(3, C31, P1)-C31-P1
STATE:
      P1:
            pending:={3}
            status:=waiting
      P2:
            pending:={2}
            status:=waiting
      P3:
            pending:={3}
            status:=waiting
      C12:
            buffer:={}
      C21:
            buffer:={}
      C23:
            buffer:={}
      C32:
            buffer:={}
      C31:
            buffer:={}
      C13:
            buffer:={}
TRANSITION:
      send(3, P1, C12)-P1-C12
STATE:
      P1:
            pending:={3}
            status:=waiting
      P2:
            pending:={2}
            status:=waiting
      P3:
            pending:={3}
            status:=waiting
      C12:
            buffer:={3}
```

```
      C21:
            buffer:={}
      C23:
            buffer:={}
      C32:
            buffer:={}
      C31:
            buffer:={}
      C13:
            buffer:={}
TRANSITION:
      receive(3, C12, P2)-C12-P2
STATE:
      P1:
            pending:={3}
            status:=waiting
      P2:
            pending:={3}
            status:=waiting
      P3:
            pending:={3}
            status:=waiting
      C12:
            buffer:={}
      C21:
            buffer:={}
      C23:
            buffer:={}
      C32:
            buffer:={}
      C31:
            buffer:={}
      C13:
            buffer:={}
TRANSITION:
      send(3, P2, C23)-P2-C23
STATE:
      P1:
            pending:={3}
            status:=waiting
      P2:
            pending:={3}
            status:=waiting
      P3:
            pending:={3}
            status:=waiting
      C12:
            buffer:={}
      C21:
            buffer:={}
      C23:
            buffer:={3}
      C32:
            buffer:={}
      C31:
            buffer:={}
      C13:
            buffer:={}
```

```
TRANSITION:
      receive(3, C23, P3)-C23-P3
STATE:
      P1:
            pending:={3}
            status:=waiting
      P2:
            pending:={3}
            status:=waiting
      P3:
            pending:={3}
            status:=elected
      C12:
            buffer:={}
      C21:
            buffer:={}
      C23:
            buffer:={}
      C32:
            buffer:={}
      C31:
            buffer:={}
      C13:
            buffer:={}
```

Appendix 2: Code for DAVE

DAVE.java

```
import java.awt.*;
import java.awt.datatransfer.*;
import java.awt.event.*;
import java.io.*;
//import java.util.Vector;
import java.util.*;
import java.beans.*;
import ProcessBean.*;
import Variable.*;
import Transition.*;



public class DAVE extends Frame{
public static void main(String[] args)
{
      new DAVE();
}

  DAVEDisplay disp;
  ScrollPane pane;
  Panel toolbar;
  Player player;
  Thread playthread;
  Button newbutton;
  Button configbutton;
  Button connectbutton;
  Button loadbutton;
  Button execbutton;
  Button nextbutton;
  Button prevbutton;
  Button stopbutton;
  Button playbutton;
  Button pausebutton;
  GridBagLayout flow;
  GridBagConstraints gbc;
  boolean playing = false;
  public DAVE() {
    super("DAVE");
    this.setBackground(Color.lightGray);
    flow = new GridBagLayout();
```

```
    flow.columnWidths = new int[] {35, 35, 35, 35, 35, 35, 35, 35, 35,
35, 35, 35, 35, 35, 35, 35, 35, 35, 35, 35};
    flow.rowHeights = new int[] {35, 700};
    gbc = new GridBagConstraints();
    newbutton = new Button("New process");
    newbutton.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {new_process();}});
    newbutton.setSize(80, 30);
    configbutton = new Button("Configure");
    configbutton.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {configure();}});
    configbutton.setSize(80, 30);
    connectbutton = new Button("Connect");
    connectbutton.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {connect();}});
    connectbutton.setSize(80, 30);
    loadbutton = new Button("Load Algorithm");
    loadbutton.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {load_state();}});
    loadbutton.setSize(80, 30);
    execbutton = new Button("Load Execution");
    execbutton.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {load_exec();}});
    execbutton.setSize(80, 30);
    nextbutton = new Button(">");
    nextbutton.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {step();}});
    nextbutton.setSize(30, 30);
    prevbutton = new Button("<");
    prevbutton.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {stepback();}});
    prevbutton.setSize(30, 30);
    stopbutton = new Button("STOP");
    stopbutton.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {stop();}});
    stopbutton.setSize(30, 30);
    playbutton = new Button("PLAY");
    playbutton.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {play();}});
    playbutton.setSize(30, 30);
    pausebutton = new Button("||");
    pausebutton.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {pause();}});
    pausebutton.setSize(30, 30);
    pane = new ScrollPane();
    pane.setSize(700, 700);
    this.setLayout(flow);
    gbc.gridwidth = 1;
    gbc.gridheight = 1;
    gbc.gridx = 0;
    gbc.gridy = 0;
    this.add(newbutton, gbc);
    gbc.gridx = 1;
    gbc.gridy = 0;
    this.add(configbutton, gbc);
    gbc.gridx = 2;
    gbc.gridy = 0;
    this.add(connectbutton, gbc);
    gbc.gridx = 3;
    gbc.gridy = 0;
    this.add(loadbutton, gbc);
    gbc.gridx = 4;
    gbc.gridy = 0;
    this.add(execbutton, gbc);
```

```
    gbc.gridx = 5;
    gbc.gridy = 0;
    this.add(prevbutton, gbc);
    gbc.gridx = 6;
    gbc.gridy = 0;
    this.add(playbutton, gbc);
    gbc.gridx = 7;
    gbc.gridy = 0;
    this.add(pausebutton, gbc);
    gbc.gridx = 8;
    gbc.gridy = 0;
    this.add(stopbutton, gbc);
    gbc.gridx = 9;
    gbc.gridy = 0;
    this.add(nextbutton, gbc);
    gbc.gridx = 0;
    gbc.gridy = 1;
    gbc.gridheight = 1;
    gbc.gridwidth = 20;
    this.add(pane, gbc);
    disp = new DAVEDisplay(this, 10000, 3600, pane);
    pane.add(disp);
    pane.setBackground(Color.white);
    disp.setBackground(Color.white);
    MenuBar menubar = new MenuBar();
    this.setMenuBar(menubar);
    Menu sim = new Menu("Process");
      Menu configuration = new Menu("Configuration");
      Menu playback = new Menu("Playback");
    menubar.add(sim);
      menubar.add(playback);
      menubar.add(configuration);
    MenuItem p, c, l, g, a, e, save, load, faster, slower, global,
delete;
    sim.add(p = new MenuItem("New process", new
MenuShortcut(KeyEvent.VK_P)));
      sim.add(delete = new MenuItem("Remove Process", new
MenuShortcut(KeyEvent.VK_R)));
    sim.add(c = new MenuItem("Connect", new
MenuShortcut(KeyEvent.VK_C)));
    sim.add(l = new MenuItem("Load Algorithm", new
MenuShortcut(KeyEvent.VK_L)));
    sim.add(g = new MenuItem("Configure process", new
MenuShortcut(KeyEvent.VK_G)));
      sim.add(global = new MenuItem("Preferences", new
MenuShortcut(KeyEvent.VK_G)));
    sim.add(e = new MenuItem("Load Execution", new
MenuShortcut(KeyEvent.VK_E)));
    configuration.add(save = new MenuItem("Save Configuration", new
MenuShortcut(KeyEvent.VK_S)));
    configuration.add(load = new MenuItem("Load Configuration", new
MenuShortcut(KeyEvent.VK_O)));
      playback.add(faster = new MenuItem("Faster", new
MenuShortcut(KeyEvent.VK_F)));
      playback.add(slower = new MenuItem("Slower", new
MenuShortcut(KeyEvent.VK_D)));
    p.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {new_process();}});
      delete.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {remove_process();}});
    c.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {connect();}});
    l.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {load_state();}});
```

```
    g.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {configure();}});
      global.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {global_configure();}});
    e.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {load_exec();}});
    save.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {save_config();}});
    load.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {load_config();}});
      faster.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {faster();}});
      slower.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {slower();}});
    this.addWindowListener(new WindowAdapter() {public void
windowClosing(WindowEvent e) {close();}});
    this.setSize(730, 730);
      player = new Player(disp);
      playthread = new Thread(player);
    this.pack();
    this.show();
      playthread.start();
  }//main

  void new_process()
  {
    Point pos = pane.getScrollPosition();
    Dimension d = pane.getViewportSize();
    int newx = pos.x + d.width/2;
    int newy = pos.y + d.height/2;
    String newname = "New";
      Color newcolor = Color.blue;
    float newfactor = (float)1.0;
    disp.new_process(newname, newcolor, newy, newx, newfactor);
  }

  void remove_process()
  {
        disp.delete();
  }

  void faster()
  {
        player.increment();
  }

  void slower()
  {
        player.decrement();
  }

  void configure()
  {
      disp.configure();
  }

  void global_configure()
  {
      disp.global_configure();
  }


  void connect()
  {
```

```
    System.out.println("Processing connect button event.");
    disp.connect();
}

void save_config()
{
    disp.save_config_create();
}

void load_config()
{
    disp.load_config_create();
}

void load_state()
{
    disp.loadCreate();
}

void load_exec()
{
    disp.loadExecCreate();
}

void step()
{
    disp.next();
}

void stop()
{
    disp.pause();
    disp.reset();
    disp.init();
}

void stepback()
{
    disp.previous();
}

void play()
{
    disp.play();
}

void pause()
{
    disp.pause();
}

void movepane(int factor)
{
      Point current =  pane.getScrollPosition();
    current.x = factor * current.x;
    current.y = factor * current.y;
    pane.setScrollPosition(current);
      repaint();
}

void movepaneout(int factor)
{
      Point current =  pane.getScrollPosition();
    current.x = current.x/factor;
```

```
        current.y = current.y/factor;
        pane.setScrollPosition(current);
          repaint();
    }


    void close()
    {
        System.exit(0);
    }

}//BenchTest

// ----- Parts -----

class DAVEDisplay extends Component implements ActionListener{
    Vector processes = new Vector();
    ProcessBean currentbean = new ProcessBean("invalid");
    ProcessBean connectbean = new ProcessBean("invalid");
    int width;
    int height;
    int globalsize = 12;
    Color color1 = Color.blue;
    Color color2 = Color.green;
    boolean attachmode = false;
    float scale;
    Frame frame;
    PopupMenu myPopup;
    PopupMenu connectPopup;
    Dialog myConfig;
    Dialog myglobalConfig;
    FileDialog myLoad;
    GridLayout myLayout;
    GridLayout myglobalLayout;
    Menu myVisibleMenu;
    Vector myVisibleMenuItems;
    int myLastX, myLastY;
    ScrollPane myScrollParent;
    boolean myMouseDown;
    boolean myMouseUp;
    boolean myMouseDrag;
    boolean play = false;
    boolean processing = false;
    Button okbutton;
    Button cancelbutton;
    Button globalokbutton;
    Button globalcancelbutton;
    Choice colorchoice;
    Choice color1choice;
    Choice color2choice;
    Choice fontchoice;
    TextField nametext;
    TextField heighttext;
    TextField widthtext;
    TextField sizetext;
    Label widthlabel;
    Label heightlabel;
    Label namelabel;
    Label colorlabel;
    Label fontlabel;
    Label sizelabel;
    Label color1label;
    Label color2label;
    Dialog myViewer;
```

```
  Button viewok;
  TextArea viewertext;
  GridBagConstraints gbc;
  boolean connectmode = false;
  Vector exectext = new Vector();
  int execpointer = 0;

  public DAVEDisplay(Frame frame, int width, int height, ScrollPane p) {
      this.frame = frame;
      this.width = width;
        this.height = height;
      this.scale = 1;
      this.myScrollParent = p;
      myVisibleMenuItems = new Vector();
      this.enableEvents(AWTEvent.MOUSE_EVENT_MASK);
      PopupMenuCreate();
      }

  public Dimension getPreferredSize() {return new Dimension(10000,
1200);}

  public void delete()
  {
        Enumeration enum = processes.elements();
        while(enum.hasMoreElements())
        {
              ProcessBean pb = (ProcessBean)enum.nextElement();
              pb.delete(currentbean);
        }
        currentbean.unmake_current();
        processes.removeElement(currentbean);
        repaint();
  }

  public void play()
  {
        play = true;
  }

  public void pause()
  {
        play = false;
  }

  public boolean isplaying()
  {
        return play;
  }

  public void new_process(String name, Color c, int y, int x, float f)
  {
      ProcessBean newbean = new ProcessBean(name, c, y, x, f);
      newbean.setFontSize(globalsize);
        if(!(currentbean.getName()).equals("invalid"))
      {
            currentbean.unmake_current();
      }
        processes.addElement((Object)newbean);
        currentbean = newbean;
        currentbean.make_current();
        repaint();
  }

  public void connect()
```

```
    {
        System.out.println("Processing internal connect command.");
        if(connectmode == false)
        {
              connectmode = true;
              System.out.println("In connectmode.");

        this.setCursor(Cursor.getPredefinedCursor(Cursor.CROSSHAIR_CURSOR)
);
        }
    }

    public void configure()
    {
        configCreate();
        myConfig.pack();
        myConfig.show();
    }

    public void global_configure()
    {
        globalConfigCreate();
        myglobalConfig.pack();
        myglobalConfig.show();
    }


    public Color texttocolor(String t)
    {
        if(t.equals("Black"))
        {
              return Color.black;
        }
        if(t.equals("Red"))
        {
              return Color.red;
        }
        if(t.equals("Blue"))
        {
              return Color.blue;
        }
        if(t.equals("Yellow"))
        {
              return Color.yellow;
        }
        if(t.equals("Green"))
        {
              return Color.green;
        }
        if(t.equals("White"))
        {
              return Color.white;
        }
        return Color.white;
    }

    public void actionPerformed(ActionEvent event)
    {
          String command = event.getActionCommand();
          if(command.equals("configure"))
          {
              configCreate();
              myConfig.pack();
              myConfig.show();
```

```
        }
        else
        if (command.equals("OK")) {
            currentbean.setName(nametext.getText());

      currentbean.setColor(texttocolor(colorchoice.getSelectedItem()));
            currentbean.setSize(Integer.parseInt(widthtext.getText()),
Integer.parseInt(heighttext.getText()));
            currentbean.setFont(fontchoice.getSelectedItem());
            myConfig.hide();
            repaint();
        }
        else
        if (command.equals("Cancel")) {
            myConfig.hide();
            repaint();
        }
        else
        if (command.equals("globalOK")) {
            Enumeration enum = processes.elements();
            while(enum.hasMoreElements())
            {
                  ProcessBean pb = (ProcessBean)enum.nextElement();

      pb.setColor1(texttocolor(color1choice.getSelectedItem()));

      pb.setColor2(texttocolor(color2choice.getSelectedItem()));
                  pb.setFontSize(Integer.parseInt(sizetext.getText()));
            }
            color1 = texttocolor(color1choice.getSelectedItem());
            color2 = texttocolor(color2choice.getSelectedItem());
            globalsize = Integer.parseInt(sizetext.getText());
            myglobalConfig.hide();
            repaint();
        }
        else
        if (command.equals("globalCancel")) {
            myglobalConfig.hide();
            repaint();
        }
        else
        {
            Enumeration enum = connectbean.getTransitions();
            System.out.println("Processing connect command.");
            while(enum.hasMoreElements())
            {
                  Transition t1 = (Transition)enum.nextElement();
                  if(command.equals(t1.getProperName()))
                  {
                        System.out.println("Found correct connecting
transition.");
                        currentbean.connect2(connectbean, t1);
                  }
            }
            processing = false;
            repaint();
        }
      return;
  }

  public void checkcurrent(int x, int y)
  {
      Enumeration enum = processes.elements();
      ProcessBean pb;
```

```
        boolean gothit;
          while(enum.hasMoreElements())
        {
              pb = (ProcessBean)enum.nextElement();
              if(pb.hitTest(x, y))
              {
                    if(!(currentbean.getName()).equals("invalid"))
                    {
                          currentbean.unmake_current();
                    }
                    currentbean = pb;
                    currentbean.make_current();
              }
        }
  }

  public void DisplayState(ProcessBean pb)
  {
        myViewer = new Dialog(frame, pb.getName());
        gbc = new GridBagConstraints();
        GridBagLayout gbl = new GridBagLayout();
          gbl.columnWidths = new int[] {300};
        gbl.rowHeights = new int[] {300, 50};
        gbc.gridx = 0;
        gbc.gridy = 0;
          gbc.gridheight = 1;
        gbc.gridwidth = 1;
        String viewtext = "State:";
        Enumeration enum = pb.getVariables();
        int maxwidth = 100;
        int columns = 0;
        while(enum.hasMoreElements())
        {
              Variable v = (Variable)enum.nextElement();
              viewtext = viewtext + "\n" + v.getName() + ":" +
v.getValue();
              columns += 1;
              if(viewtext.length() > maxwidth)
              {
                    maxwidth = viewtext.length();
              }
        }
        enum = pb.getTransitions();
        viewtext = viewtext + "\n" + "Transitions:";
        while(enum.hasMoreElements())
        {
              Transition t = (Transition)enum.nextElement();
              viewtext = viewtext + "\n" + t.getName();
              columns += 1;
              if(viewtext.length() > maxwidth)
              {
                    maxwidth = viewtext.length();
              }
        }
        if(columns<30)
        {
              columns = 30;
        }
        viewertext = new TextArea(viewtext, columns, maxwidth,
TextArea.SCROLLBARS_BOTH);
        viewertext.setSize(280,280);
        viewok = new Button("OK");
          viewok.setSize(80, 30);
        viewok.setActionCommand("ViewOK");
```

```
        viewok.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {viewerok();}});
      myViewer.setLayout(gbl);
      gbc.gridx = 0;
      gbc.gridy = 0;
      myViewer.add(viewertext, gbc);
      gbc.gridx = 0;
      gbc.gridy = 1;
      myViewer.add(viewok, gbc);
      myViewer.pack();
      myViewer.show();
  }

  public void viewerok()
  {
      myViewer.hide();
  }

  public void DisplayVariable(Variable v)
  {
      myViewer = new Dialog(frame, v.getName());
      gbc = new GridBagConstraints();
      GridBagLayout gbl = new GridBagLayout();
        gbl.columnWidths = new int[] {300};
      gbl.rowHeights = new int[] {300, 50};
      gbc.gridx = 0;
      gbc.gridy = 0;
        gbc.gridheight = 1;
      gbc.gridwidth = 1;
      String viewtext = "";
      int maxwidth = 280;
      int columns = 30;
      viewtext = viewtext + "\n" + v.getName() + ":" + v.getValue();
        maxwidth = viewtext.length();
      viewertext = new TextArea(viewtext, columns, maxwidth,
TextArea.SCROLLBARS_BOTH);
      gbc.gridx = 0;
      gbc.gridy = 1;
      viewok = new Button("OK");
      viewok.setActionCommand("ViewOK");
        viewok.addActionListener(new ActionListener() {public void
actionPerformed(ActionEvent e) {viewerok();}});
        myViewer.setLayout(gbl);
      gbc.gridx = 0;
      gbc.gridy = 0;
      myViewer.add(viewertext, gbc);
      gbc.gridx = 0;
      gbc.gridy = 1;
      myViewer.add(viewok, gbc);
      myViewer.pack();
      myViewer.show();
  }

  public boolean checkhotspot(int x, int y)
  {
      Enumeration enum = processes.elements();
      ProcessBean pb;
      while(enum.hasMoreElements())
      {
            pb = (ProcessBean)enum.nextElement();
            if(pb.isShorthand())
            {
                  if(pb.inSpot(x, y))
                  {
```

```
                        DisplayState(pb);
                        return true;
                  }
            }
            Enumeration enum2 = pb.getVariables();
            while(enum2.hasMoreElements())
            {
                  Variable v = (Variable)enum2.nextElement();
                  if(v.inSpot(x, y))
                  {
                        DisplayVariable(v);
                        return true;
                  }
            }
      }
      return false;
}
public void checkconnect(int x, int y)
{
      Enumeration enum = processes.elements();
      ProcessBean pb;
      ProcessBean hit = currentbean;
      boolean foundhit = false;
      Vector candidates = new Vector();
      boolean success = false;
      System.out.println("Attempting to connect.");
      processing = true;
      while(enum.hasMoreElements())
      {
            pb = (ProcessBean)enum.nextElement();
            if(pb.hitTest(x, y))
            {
                  System.out.println("Found proper bean to connect.");
                  hit = pb;
                  foundhit = true;
                  //success = currentbean.connect(pb, x, y);
            }
      }
      if(!foundhit)
      {
            return;
      }
      Enumeration enum2 = hit.getTransitions();
      while(enum2.hasMoreElements())
      {
            Transition tdest = (Transition)enum2.nextElement();
            Enumeration enum3 = currentbean.getTransitions();
            while(enum3.hasMoreElements())
            {
                  Transition tsource = (Transition)enum3.nextElement();

      if((tsource.getProperName().equals(tdest.getProperName())) &&
                     (tsource.getType().equals("output")) &&
                     (tdest.getType().equals("input")))
                  {
                        candidates.addElement(tdest);
                  }
            }
      }
      connectbean = hit;
      if(ConnectMenuCreate(candidates))
      {
           connectPopup.show(this, myLastX, myLastY);
   }
```

```
      this.setCursor(Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR));
    connectmode = false;
      return;
  }

  public void processMouseEvent(MouseEvent e) {
        if(processing)
        {
              return;
        }
        if (e.isPopupTrigger()) {
              myLastX = e.getX();
              myLastY = e.getY();
              checkcurrent(myLastX, myLastY);
              myPopup.show(this, myLastX, myLastY);
        }
        else
        if ((e.getID() == MouseEvent.MOUSE_CLICKED) &&
(!e.isPopupTrigger()))
        {
              myLastX = e.getX();
              myLastY = e.getY();
              boolean hotcheck = false;
              if(!connectmode)
              {
                  checkcurrent(myLastX, myLastY);
              }
              else
              {
                  if((myLastX >= (currentbean.getAttachx() - 2)) &&
(myLastX <= (currentbean.getAttachx() + 2))
                        && (myLastY >= (currentbean.getAttachy() - 2))
&& (myLastY <= (currentbean.getAttachy() + 2)))
                  {
                        System.out.println("Going into attachmode.");
                        attachmode = true;
                        return;
                  }
                  hotcheck = checkhotspot(myLastX, myLastY);
                  if(hotcheck)
                  {
                        return;
                  }
                  System.out.println("Mouse says to connect.");
                  checkconnect(myLastX, myLastY);
              }
              myMouseDown = true;
              System.out.println("Click");
              repaint();
        }
        else
        if((e.getID() == MouseEvent.MOUSE_PRESSED) && (!(e.getID() ==
MouseEvent.MOUSE_CLICKED)))
        {
              myLastX = e.getX();
              myLastY = e.getY();
          if((myLastX >= (currentbean.getAttachx() - 2)) && (myLastX <=
(currentbean.getAttachx() + 2))
                  && (myLastY >= (currentbean.getAttachy() - 2)) &&
(myLastY <= (currentbean.getAttachy() + 2)))
              {
                    System.out.println("Going into attachmode II.");
                      attachmode = true;
                      return;
```

```
                }
                else
                {
                    myMouseDrag = true;
                }
        }
        else
        if(e.getID() == MouseEvent.MOUSE_RELEASED)
        {
                int diffx = (e.getX()) - myLastX;
                int diffy = (e.getY()) - myLastY;
                boolean hotcheck = false;
                if(diffx < 0)
                {
                    diffx = -1 * diffx;
                }
                if(diffy < 0)
                {
                    diffy = -1 * diffy;
                }
                if(connectmode)
                {
                    checkconnect(e.getX(), e.getY());
                    System.out.println("Frag 1: Mouse also says to
connect.");
                }
                else
                {
                    hotcheck = checkhotspot(e.getX(), e.getY());
                    if(hotcheck)
                    {
                          return;
                    }
                    else
                    if(attachmode)
                    {
                          System.out.println("Exiting attachmode.");
                          currentbean.setAttach(e.getX(), e.getY());
                          attachmode = false;
                          repaint();
                    }
                    else
                    if((diffx > 5) && (diffy>5))
                      {
                          myLastX = e.getX();
                                  myLastY = e.getY();
                          if(myMouseDrag)
                          {
                                currentbean.setpos(myLastX, myLastY);
                                System.out.println("Dragged");
                                repaint();
                          }
                          myMouseDrag = false;
                      }
                 }
        }
        else super.processMouseEvent(e);
  }

  public void paint(Graphics g)
  {
      //Dimension size = myScrollParent.getViewportSize();
      //Point corner = myScrollParent.getScrollPosition();
      //g.setClip(corner.x, corner.y, size.width, size.height);
```

```
        System.out.println("Repainting");
        Enumeration enum = processes.elements();
        ProcessBean pb;
        while(enum.hasMoreElements())
        {
              pb = (ProcessBean)enum.nextElement();
              pb.setpos(pb.getX(), pb.getY());
              pb.paint(g);
        }
    }

    public void PopupMenuCreate() {
          myPopup = new PopupMenu();
          MenuItem mi = new MenuItem("Configure");
          mi.setActionCommand("configure");
          mi.addActionListener(this);
          myPopup.add(mi);
          MenuItem item = new MenuItem("Connect");
          item.setActionCommand("connect");
          item.addActionListener(this);
          myPopup.add(item);
          this.add(myPopup);
    }

    public boolean ConnectMenuCreate(Vector candidates) {
          connectPopup = new PopupMenu();
          Enumeration enum = candidates.elements();
          boolean hasEls = false;
          while(enum.hasMoreElements())
          {
                hasEls = true;
                Transition t = (Transition)enum.nextElement();
                System.out.println("Menu option: " + t.getProperName());
                MenuItem mi = new MenuItem(t.getProperName());
                mi.setActionCommand(t.getProperName());
                mi.addActionListener(this);
                connectPopup.add(mi);
          }
          this.add(connectPopup);
          return hasEls;
    }


    public void loadCreate()
    {
        myLoad = new FileDialog(frame, "Load State", FileDialog.LOAD);
        myLoad.setDirectory(".");
        myLoad.show();
        currentbean.loadState(new File(myLoad.getFile()));
        repaint();
    }

    public void load_config_create()
    {
        myLoad = new FileDialog(frame, "Load Configuration",
FileDialog.LOAD);
        myLoad.setDirectory(".");
        myLoad.setFilenameFilter(new cfgFilter());
        myLoad.show();
        load_config(new File(myLoad.getFile()));
        repaint();
    }

    public void save_config_create()
```

```
  {
      myLoad = new FileDialog(frame, "Save Configuration",
FileDialog.LOAD);
      myLoad.setDirectory(".");
      myLoad.show();
      save_config(new File(myLoad.getFile()));
      repaint();
  }


  public void loadExecCreate()
  {
      myLoad = new FileDialog(frame, "Load Execution", FileDialog.LOAD);
      myLoad.setDirectory(".");
      myLoad.show();
      loadExecution(new File(myLoad.getFile()));
      repaint();
  }

  public Color get_color(String s)
  {
      int r = 0;
      int g = 0;
      int b = 0;
      String vals = s.substring(s.indexOf('[') + 1, s.indexOf(']'));
      String red = vals.substring(0, vals.indexOf(','));
      red = red.substring(red.indexOf('=') + 1);
      System.out.println("Red:" + red);
      vals = vals.substring(vals.indexOf(',') + 1);
      String green = vals.substring(0, vals.indexOf(','));
      green = green.substring(green.indexOf('=') + 1);
      System.out.println("Green:" + green);
      vals = vals.substring(vals.indexOf(',') + 1);
      String blue = vals;
      blue = blue.substring(blue.indexOf('=') + 1);
      System.out.println("blue:" + blue);
      try
      {
            r = (Integer.valueOf(red)).intValue();
      }
      catch(NumberFormatException e)
      {
            return(Color.black);
      }
      try
      {
            g = (Integer.valueOf(green)).intValue();
      }
      catch(NumberFormatException e)
      {
            return(Color.black);
      }
      try
      {
            b = (Integer.valueOf(blue)).intValue();
      }
      catch(NumberFormatException e)
      {
            return(Color.black);
      }
      return(new Color(r, g, b));
  }

  public void load_config(File f)
```

```
  {
      String currentline = new String("");
      char curr[] = new char[1];
      int index = 0;
      int result = 0;
      int x = 0;
      int y = 0;
      int iwidth = 0;
      int iheight = 0;
      FileReader fr;
      try
      {
            fr = new FileReader(f);
      }
      catch(FileNotFoundException e)
      {
            System.out.println("Error opening file.");
            return;
      }
      catch(IOException e)
      {
            System.out.println("File not found on load.");
            return;
      }
      processes = new Vector();
      while(result != -1)
      {
            while(curr[0] != '\n')
            {
                  try
                  {
                        result = fr.read(curr, 0, 1);
                  }
                  catch(IOException e)
                  {
                        System.out.println("Error reading file #111");
                        return;
                  }
                  if(result == -1)
                  {
                        break;
                  }
                  else
                  {
                        currentline = currentline + (new String(curr, 0,
1));
                  }
            }
            System.out.println("Beginning a new loop.");
            System.out.println(currentline);
            System.out.println("Printed currentline.");
            if(currentline.indexOf(':') < 0)
            {
                  break;
            }
            String type = currentline.substring(0,
currentline.indexOf(':'));
            System.out.println("Got past type.");
            type = type.trim();
            System.out.println(type);
            if(currentline.indexOf(':') < 0)
            {
                  break;
            }
```

```
            String data = currentline.substring(currentline.indexOf(':')
+ 1);
            System.out.println("To loop.");
            if(type.equals("PROCESS"))
            {
                  System.out.println("in loop.");
                  if(data.indexOf(';') < 0)
                  {
                        break;
                  }
                  String Name = data.substring(0, data.indexOf(';'));
                  System.out.println("Name:" + Name);
                  if(data.indexOf(';') < 0)
                  {
                        break;
                  }
                  data = data.substring(data.indexOf(';') + 1);
                  if(data.indexOf(';') < 0)
                  {
                        break;
                  }
                  String X = data.substring(0, data.indexOf(';'));
                  x = Integer.parseInt(X);
                  System.out.println("X:" + X);
                  if(data.indexOf(';') < 0)
                  {
                        break;
                  }
                  data = data.substring(data.indexOf(';') + 1);
                  if(data.indexOf(';') < 0)
                  {
                        break;
                  }
                  String Y = data.substring(0, data.indexOf(';'));
                  y = Integer.parseInt(Y);
                  System.out.println("Y:" + Y);
                  if(data.indexOf(';') < 0)
                  {
                        break;
                  }
                  data = data.substring(data.indexOf(';') + 1);
                  if(data.indexOf(';') < 0)
                  {
                        break;
                  }
                  String width = data.substring(0, data.indexOf(';'));
                  iwidth = Integer.parseInt(width);
                  System.out.println("Width:" + width);
                  if(data.indexOf(';') < 0)
                  {
                        break;
                  }
                  data = data.substring(data.indexOf(';') + 1);
                  if(data.indexOf(';') < 0)
                  {
                        break;
                  }
                  String height = data.substring(0, data.indexOf(';'));
                  iheight = Integer.parseInt(height);
                  System.out.println("Height:" + height);
                  if(data.indexOf(';') < 0)
                  {
                        break;
                  }
```

```
                    data = data.substring(data.indexOf(';') + 1);
                    if(data.indexOf(';') < 0)
                    {
                          break;
                    }
                    String color = data.substring(0, data.indexOf(';'));
                    System.out.println("Color:" + color);
                    Color newColor = get_color(color);
                    if(data.indexOf(';') < 0)
                    {
                          break;
                    }
                    data = data.substring(data.indexOf(';') + 1);
                    if(data.indexOf(';') < 0)
                    {
                          break;
                    }
                    String font = data.substring(0, data.indexOf(';'));
                    System.out.println("Font:" + font);
                    if(data.indexOf(';') < 0)
                    {
                          break;
                    }
                    data = data.substring(data.indexOf(';') + 1);
                    if(data.indexOf(';') < 0)
                    {
                          break;
                    }
                    String file = data.substring(0, data.indexOf(';'));
                    System.out.println("File:" + file);
                    ProcessBean p = new ProcessBean(Name, newColor, y, x,
(float)1.0);
                    p.setFontSize(globalsize);
                    p.setSize(iwidth, iheight);
                    p.setColor(newColor);
                    File newFile;
                    newFile = new File(file);
                    p.loadState(newFile);
                    processes.addElement(p);
                    System.out.println("Done!");
              }
              if(type.equals("COMPOSITION"))
              {
                    repaint();
                    String Name = data;
                    System.out.println("Configuration:");
                    String base = data.substring(0, data.indexOf('-'));
                    data = data.substring(data.indexOf('-')+1);
                    String P1 = data.substring(0, data.indexOf('-'));
                    System.out.println("Source: " + P1);
                    data = data.substring(data.indexOf('-') + 1);
                    String P2 = data.substring(0, data.indexOf(';'));
                    P1 = P1.trim();
                    P2 = P2.trim();
                    System.out.println("Destination: " + P2);
                    Enumeration enum = processes.elements();
                    ProcessBean pb1 = (ProcessBean)enum.nextElement();
                    ProcessBean pb2 = (ProcessBean)enum.nextElement();
                    enum = processes.elements();
                    while(enum.hasMoreElements())
                    {
                          ProcessBean pb =
(ProcessBean)enum.nextElement();
                          if(pb.getName().equals(P1))
```

```
                        {
                              pb1 = pb;
                              System.out.println("Found correct source:"
+ pb1.getName());
                        }
                        if(pb.getName().equals(P2))
                        {
                              pb2 = pb;
                              System.out.println("Found correct
destination:" + pb2.getName());
                        }
                  }
                  if(true)
                  {
                        Enumeration enum2 = pb2.getTransitions();
                        String base2 = base.substring(0,
base.indexOf('('));
                        Transition t = (Transition)enum2.nextElement();
                        enum2 = pb2.getTransitions();
                        while(enum2.hasMoreElements())
                        {
                              t = (Transition)enum2.nextElement();
                              if(t.getProperName().equals(base2))
                              {
                                    System.out.println("Found correct
transition:" + t.getName());
                                    x = t.getX() + 1;
                                    y = t.getY() - 1;
                                    break;
                              }
                        }
                        pb1.connect2(pb2, t);
                  }
            }
            curr[0] = ' ';
            currentline = "";
            x = 0;
            y = 0;
      }
  }

  public void save_config(File f)
  {
      String currentline = new String("");
      int index = 0;
      FileWriter fw;
      try
      {
            fw = new FileWriter(f);
      }
      catch(FileNotFoundException e)
      {
            System.out.println("Failed to save configuration.");
            return;
      }
      catch(IOException e)
      {
            System.out.println("IO Exception on save.");
            return;

      }
      Enumeration enum = processes.elements();
      while(enum.hasMoreElements())
      {
```

```
            ProcessBean pb = (ProcessBean)enum.nextElement();
            currentline = "PROCESS:" + pb.getName() + ";";
            currentline += pb.getX() + ";" + pb.getY() + ";" +
pb.getWidth() + ";" + pb.getHeight() + ";";
            currentline += pb.getColor().toString() + ";" +
pb.getFontName() + ";" + pb.getFile().getName() + ";\n\r";
            try
            {
                  fw.write(currentline, 0, currentline.length());
            }
            catch(IOException e)
            {
                  System.out.println("IO Exception on pb save.");
                  return;
            }
            currentline = "";
      }
      enum = processes.elements();
      while(enum.hasMoreElements())
      {
            ProcessBean pb = (ProcessBean)enum.nextElement();
            Enumeration enum2 = pb.connections();
            while(enum2.hasMoreElements())
            {
                  Connection c = (Connection)enum2.nextElement();
                  Enumeration enum3 = c.getCompositions();
                  while(enum3.hasMoreElements())
                  {
                        Composition comp =
(Composition)enum3.nextElement();
                        currentline = "COMPOSITION:" + comp.getName() +
";\n\r";
                        try
                        {
                              fw.write(currentline, 0,
currentline.length());
                        }
                        catch(IOException e)
                        {
                              System.out.println("IO Exception on comp
save.");
                              return;
                        }
                  }
            }
      }
      try
      {
            fw.flush();
      }
      catch(IOException e)
      {
            System.out.println("IO Exception on flush.");
      }
      try
      {
            fw.close();
      }
      catch(IOException e)
      {
            System.out.println("IO Exception on flush.");
      }

  }
```

```
  public void loadExecution(File algfile)
  {
      char nextc[] = new char[1];
      String currentline = new String("");
      int index = 0;
      FileReader fr;
      try{
            fr = new FileReader(algfile);
      }
      catch(FileNotFoundException e)
      {
            System.out.println("Shit!");
            return;
      }
      while(index != -1)
      {
            try
            {
                  index = fr.read(nextc, 0, 1);
            }
            catch(IOException e)
            {
                  break;
            }
            if(nextc[0] == '\n')
            {
                  exectext.addElement(currentline);
                  currentline = new String("");
            }
            else
            {
                  if(index != -1)
                  {
                        currentline = currentline.concat((new
Character(nextc[0])).toString());
                  }
            }
            if(index == -1)
            {
                  exectext.addElement(currentline);
                  currentline = new String("");
            }
      }
      init();
  }


  public void init()
  {
      String varname;
      String procname;
      String value;
      String process;
      execpointer = 0;
      String line = (String)exectext.elementAt(execpointer);
      if(!(line.equals("STATE:")))
      {
            execerror();
      }
      while(!line.equals("TRANSITION:") && (execpointer <
exectext.size()))
      {
            execpointer += 1;
```

```
            line = ((String)exectext.elementAt(execpointer)).trim();
            procname = line.substring(0, line.indexOf(':'));
            execpointer += 1;
            line = ((String)exectext.elementAt(execpointer)).trim();
            while(!(line.endsWith(":")) && execpointer <
exectext.size())
            {
                  varname = line.substring(0, line.indexOf(':'));
                  value = line.substring(line.indexOf('=') + 1);
                  System.out.println("Setting a variable:" + varname +
value);
                  SetVarValue(procname, varname, value);
                  execpointer += 1;
                  if(execpointer >= exectext.size())
                  {
                        break;
                  }
                  else
                  {
                        line =
((String)exectext.elementAt(execpointer)).trim();
                  }
            }
            execpointer -= 1;
      }
      repaint();
  }

  public boolean inBounds()
  {
        if(execpointer >= exectext.size())
        {
              return true;
        }
        else
        {
              return false;
        }
  }

  public void next()
  {
      execpointer += 1;
      if(execpointer > exectext.size())
      {
            return;
      }
      String transname;
      String sourcename;
      String destname;
      String args;
      String varname;
      String procname;
      String value;
      String process;
      String tempstr;
      int temp;
      reset();
      String line = (String)exectext.elementAt(execpointer);
      if(!(line.equals("TRANSITION:")))
      {
            execerror();
      }
      while(!line.equals("STATE:") && execpointer<exectext.size())
```

```
      {
            if(execpointer < (exectext.size() - 1))
            {
                  execpointer += 1;
            }
            else
            {
                  return;
            }
            line = ((String)exectext.elementAt(execpointer)).trim();
            while(!(line.endsWith(":")) && execpointer <
(exectext.size() - 1))
            {
                  transname = line.substring(0, line.indexOf('('));
                  System.out.println("Transname." + transname);
                  value = line.substring(line.indexOf('(') + 1,
line.indexOf(')'));
                  System.out.println("value." + value);
                  temp = line.indexOf('-');
                  tempstr = line.substring(temp + 1);
                  System.out.println("tempstr:" + tempstr);
                  sourcename = tempstr.substring(0, tempstr.indexOf('-
'));
                  System.out.println("sourcename:" + sourcename);
                  destname = tempstr.substring(tempstr.indexOf('-') +
1);
                  System.out.println("Destname: " + destname);
                  System.out.println(sourcename + " " + destname + " " +
transname + " " + value);
                  SetTransValue(sourcename, destname, transname, value);
                  if(execpointer < (exectext.size() - 1))
                  {
                        execpointer += 1;
                  }
                  else
                  {
                        return;
                  }
                  line =
((String)exectext.elementAt(execpointer)).trim();
            }
      }
      while(!line.equals("TRANSITION:") && execpointer<exectext.size())
      {
            if(execpointer < (exectext.size() - 1))
            {
                  if(execpointer < (exectext.size() - 1))
                  {
                        execpointer += 1;
                  }
                  else
                  {
                        return;
                  }
                  line =
((String)exectext.elementAt(execpointer)).trim();
                  procname = line.substring(0, line.indexOf(':'));
                  if(execpointer >= exectext.size())
                  {
                        break;
                  }
                  else
                  {
                        if(execpointer < (exectext.size() - 1))
```

```
                        {
                              execpointer += 1;
                        }
                        else
                        {
                              return;
                        }
                        line =
((String)exectext.elementAt(execpointer)).trim();
                        while(!(line.endsWith(":")) &&
execpointer<exectext.size())
                        {
                              varname = line.substring(0,
line.indexOf(':'));
                              value = line.substring(line.indexOf('=') +
1);
                              System.out.println("Var: " + varname + " "
+ value);
                              SetVarValue(procname, varname, value);
                              if(execpointer < exectext.size())
                              {
                                    execpointer += 1;
                              }
                              else
                              {
                                    return;
                              }
                              if(execpointer>= (exectext.size() - 1))
                              {
                                    break;
                              }
                              else
                              {
                                    line =
((String)exectext.elementAt(execpointer)).trim();
                              }
                        }
                  }
                  if(execpointer < exectext.size())
                  {
                        execpointer -= 1;
                  }
            }
      }
      repaint();

  }

  public void previous()
  {
      int transcount = 0;
      int statecount = 0;
      execpointer -= 1;
      String current = (String)exectext.elementAt(execpointer);
      reset();
      while((transcount<2) && (statecount<2))
      {
            current = (String)exectext.elementAt(execpointer);
            if(current.equals("STATE:"))
            {
                  statecount += 1;
            }
            if(current.equals("TRANSITION:"))
            {
```

```
                    transcount += 1;
                    if(transcount == 2)
                    {
                          break;
                    }
              }
              execpointer -= 1;
              if(execpointer < 0)
              {
                    execpointer = 0;
                    init();
                    return;
              }
        }
        next();
  }

  public void reset()
  {
        Enumeration enum = processes.elements();
        while(enum.hasMoreElements())
        {
              ProcessBean pb = (ProcessBean)enum.nextElement();
              pb.reset();
        }
  }

  public void SetVarValue(String process, String varname, String value)
  {
        Enumeration enum = processes.elements();
        while(enum.hasMoreElements())
        {
              ProcessBean pb = (ProcessBean)enum.nextElement();
              if(pb.getName().equals(process))
              {
                    pb.SetVarValue(varname, value);
                    break;
              }
        }
  }

  public void SetTransValue(String source, String dest, String trans,
String value)
  {
        Enumeration enum = processes.elements();
        while(enum.hasMoreElements())
        {
              ProcessBean pb = (ProcessBean)enum.nextElement();
              if(pb.getName().equals(source))
              {
                    pb.SetTransValue(dest, trans, value);
                    break;
              }
        }
  }

  public void execerror()
  {
        return;
  }

  public void configCreate()
  {
        myConfig = new Dialog(frame, "Configure Process", false);
```

```
        nametext = new TextField(currentbean.getName(), 30);
        heighttext = new
TextField(String.valueOf(currentbean.getHeight()), 3);
        widthtext = new TextField(String.valueOf(currentbean.getWidth()),
3);
        colorchoice = new Choice();
        String colorname = "Black";
        if(currentbean.getColor().equals(Color.black))
        {
              colorname = "Black";
        }
        if(currentbean.getColor().equals(Color.red))
        {
              colorname = "Red";
        }
        if(currentbean.getColor().equals(Color.blue))
        {
              colorname = "Blue";
        }
        if(currentbean.getColor().equals(Color.yellow))
        {
              colorname = "Yellow";
        }
        if(currentbean.getColor().equals(Color.green))
        {
              colorname = "Green";
        }
        if(currentbean.getColor().equals(Color.white))
        {
              colorname = "White";
        }
        colorchoice.add("Black");
        colorchoice.add("Red");
        colorchoice.add("Blue");
        colorchoice.add("Yellow");
        colorchoice.add("Green");
        colorchoice.add("White");
        colorchoice.select(colorname);
        fontchoice = new Choice();
        fontchoice.add("TimesRoman");
        fontchoice.add("Courier");
        fontchoice.add("Helvetica");
        fontchoice.add("Dialog");
        fontchoice.add("Serif");
        fontchoice.add("SanSerif");
        fontchoice.select(currentbean.getFontName());
        okbutton = new Button("OK");
        okbutton.setActionCommand("OK");
        okbutton.addActionListener(this);
        cancelbutton = new Button("Cancel");
        cancelbutton.setActionCommand("Cancel");
        cancelbutton.addActionListener(this);
        myLayout = new GridLayout(6, 2, 2, 2);
        namelabel = new Label("Name:");
        colorlabel = new Label("Color:");
        fontlabel = new Label("Font:");
        widthlabel = new Label("Width:");
        heightlabel = new Label("Height:");
        myConfig.add(namelabel);
        myConfig.add(nametext);
        myConfig.add(colorlabel);
        myConfig.add(colorchoice);
        myConfig.add(fontlabel);
        myConfig.add(fontchoice);
```

```
    myConfig.add(widthlabel);
    myConfig.add(widthtext);
    myConfig.add(heightlabel);
    myConfig.add(heighttext);
    myConfig.add(okbutton);
    myConfig.add(cancelbutton);
    myConfig.setLayout(myLayout);
}

public void globalConfigCreate()
{
    myglobalConfig = new Dialog(frame, "Global settings", false);
    sizetext = new TextField(String.valueOf(globalsize), 3);
    color1choice = new Choice();
    color2choice = new Choice();
    String color1name = "Black";
    if(color1.equals(Color.black))
    {
          color1name = "Black";
    }
    if(color1.equals(Color.red))
    {
          color1name = "Red";
    }
    if(color1.equals(Color.blue))
    {
          color1name = "Blue";
    }
    if(color1.equals(Color.yellow))
    {
          color1name = "Yellow";
    }
    if(color1.equals(Color.green))
    {
          color1name = "Green";
    }
    if(color1.equals(Color.white))
    {
          color1name = "White";
    }
    color1choice.add("Black");
    color1choice.add("Red");
    color1choice.add("Blue");
    color1choice.add("Yellow");
    color1choice.add("Green");
    color1choice.add("White");
    color1choice.select(color1name);

    String color2name = "Black";
    if(color2.equals(Color.black))
    {
          color2name = "Black";
    }
    if(color2.equals(Color.red))
    {
          color2name = "Red";
    }
    if(color2.equals(Color.blue))
    {
          color2name = "Blue";
    }
    if(color2.equals(Color.yellow))
    {
          color2name = "Yellow";
```

```
        }
        if(color2.equals(Color.green))
        {
              color2name = "Green";
        }
        if(color2.equals(Color.white))
        {
              color2name = "White";
        }
        color2choice.add("Black");
        color2choice.add("Red");
        color2choice.add("Blue");
        color2choice.add("Yellow");
        color2choice.add("Green");
        color2choice.add("White");
        color2choice.select(color2name);

        globalokbutton = new Button("OK");
        globalokbutton.setActionCommand("globalOK");
        globalokbutton.addActionListener(this);
        globalcancelbutton = new Button("Cancel");
        globalcancelbutton.setActionCommand("globalCancel");
        globalcancelbutton.addActionListener(this);
        myglobalLayout = new GridLayout(4, 2, 2, 2);
        sizelabel = new Label("Font size:");
        color1label = new Label("Connection 1 Color:");
        color2label = new Label("Connection 2 Color:");
        myglobalConfig.add(sizelabel);
        myglobalConfig.add(sizetext);
        myglobalConfig.add(color1label);
        myglobalConfig.add(color1choice);
        myglobalConfig.add(color2label);
        myglobalConfig.add(color2choice);
        myglobalConfig.add(globalokbutton);
        myglobalConfig.add(globalcancelbutton);
        myglobalConfig.setLayout(myglobalLayout);
    }


}

class Player implements Runnable
{
        private DAVEDisplay myD;
        private double counter = 0.0;
        private double limit = 1000.00;
        public Player(DAVEDisplay d)
        {
              myD = d;
        }

        public void increment()
        {
              limit += 100.0;
        }

        public void decrement()
        {
              limit -= 100.0;
        }

        public void run()
        {
              while(true)
```

```
            {
                  if(counter > limit)
                  {
                        if(myD.isplaying())
                        {
                              if(myD.inBounds())
                              {
                                    myD.pause();
                                    counter = 0.0;
                              }
                              else
                              {
                                    myD.next();
                                    counter = 0.0;
                              }
                        }
                  }
                  if(myD.isplaying())
                  {
                        counter += 0.001;
                  }
            }
      }
}

class cfgFilter implements FilenameFilter
{
      public cfgFilter()
      {
            return;
      }

      public boolean accept(File dir, String name)
      {
            int period = name.indexOf('.');
            String extension;
            if(period > 0)
            {
                  extension = name.substring(name.indexOf('.') + 1);
            }
            else
            {
                  return false;
            }
            System.out.println("Extension: " + extension);
            if(extension.equals("cfg"))
            {
                  return true;
            }
            else
            {
                  return false;
            }
      }
}
```

ProcessBean.java

```
import java.awt.*;
import java.io.*;
import java.beans.*;
import java.util.*;
import Variable.*;
```

```
import Transition.*;
import java.lang.*;

public class ProcessBean extends Component
      implements PropertyChangeListener {

      private Vector myState = new Vector();
      private Vector myConnections = new Vector();
      private Vector myTransitions = new Vector();
      private Color myColor = Color.orange;
      private Color color1 = Color.blue;
      private Color color2 = Color.green;
      private Font myFont = new Font("Dialog", Font.PLAIN, 12);
      private String myName = "Name";
      private Rectangle myRect = new Rectangle(0, 0, 100, 100);
      private int myRadius = 100;
      private float myScaleFactor = 1;
      private int myLayer = 1;
      private boolean current = false;
      private Vector algtext = new Vector();
      private boolean shorthand = false;
      private Rectangle hotspot;
      private File af = new File("");
      private int attachx;
      private int attachy;
      boolean attachmoved = false;

      // for notifying listening beans of property changes
      private PropertyChangeSupport changes = new
PropertyChangeSupport(this);

      // constructors
      public ProcessBean() {
            double w = (double) ((((float)myRect.width)/2) *
(((float)myRect.width)/2));
            double h = (double) ((((float)myRect.height)/2) *
(((float)myRect.height)/2));
            myRadius = (int) (Math.sqrt(w + h));

      }

      public ProcessBean(String name) {
            myName = name;
            double w = (double) ((((float)myRect.width)/2) *
(((float)myRect.width)/2));
            double h = (double) ((((float)myRect.height)/2) *
(((float)myRect.height)/2));
            myRadius = (int) (Math.sqrt(w + h));

      }

      public ProcessBean(String name, Color color) {
            myName = name;
            myColor = color;
            double w = (double) ((((float)myRect.width)/2) *
(((float)myRect.width)/2));
            double h = (double) ((((float)myRect.height)/2) *
(((float)myRect.height)/2));
            myRadius = (int) (Math.sqrt(w + h));

      }

      public ProcessBean(String name, int x, int y) {
            myName = name;
```

```
            myRect.x = x;
            myRect.y = y;
            double w = (double) ((((float)myRect.width)/2) *
(((float)myRect.width)/2));
            double h = (double) ((((float)myRect.height)/2) *
(((float)myRect.height)/2));
            myRadius = (int) (Math.sqrt(w + h));

      }

      public ProcessBean(String name, Color color, int x, int y)
      {
            myName = name;
            myColor = color;
            myRect.x = x;
            myRect.y = y;
            double w = (double) ((((float)myRect.width)/2) *
(((float)myRect.width)/2));
            double h = (double) ((((float)myRect.height)/2) *
(((float)myRect.height)/2));
            myRadius = (int) (Math.sqrt(w + h));

      }

      public ProcessBean(String name, Color color, int y, int x, float
factor)
      {
            myScaleFactor = factor;
            myName = name;
            myColor = color;
            myRect.x = x;
            myRect.y = y;
            double w = (double) ((((float)myRect.width)/2) *
(((float)myRect.width)/2));
            double h = (double) ((((float)myRect.height)/2) *
(((float)myRect.height)/2));
            myRadius = (int) (Math.sqrt(w + h));
            attachx = x;
            attachy = y;
            myFont = new Font("Dialog", Font.PLAIN, (int)(myScaleFactor
* 12));
      }

      //PropertyChangeSupport methods
      public void addPropertyChangeListener(PropertyChangeListener l) {
            changes.addPropertyChangeListener(l);
      }

      public void removePropertyChangeListener(PropertyChangeListener l)
{
            changes.removePropertyChangeListener(l);
      }

      public void firePropertyChange(String propertyName, Object
oldValue,
            Object newValue) {

            changes.firePropertyChange(propertyName, oldValue,
newValue);
      }

      //PropertyChangeListener method: what to do when this is notified
      //    of a property change it's listening for
      public void propertyChange(PropertyChangeEvent evt) {
```

```
        ProcessBean pb = (ProcessBean)evt.getSource();
        //System.out.println("This is being called.");
        if (evt.getPropertyName() == "ChangeRadius")
        {
        }
        else
        {
              if (evt.getPropertyName() == "ChangeLayer")
              {
              }
        }

        //repaint();
}


// methods
public void resize(float factor)
{
        myRect.x = (int)((factor/myScaleFactor) * myRect.x);
        myRect.y = (int)((factor/myScaleFactor) * myRect.y);
        myFont = new Font("Dialog", Font.PLAIN, (int)(12 * factor));
        myScaleFactor = factor;

}

public void make_current()
{
        current = true;
}

public void unmake_current()
{
        current = false;
}

public void delete(ProcessBean pb)
{
        Enumeration enum = myConnections.elements();
        while(enum.hasMoreElements())
        {
              Connection c = (Connection)enum.nextElement();
              if(c.exists(this, pb))
              {
                    myConnections.removeElement(c);
              }
        }
}

public void loadState(File algfile)
{
        af = algfile;
        char nextc[] = new char[1];
        String currentline = new String("");
        int index = 0;
        FileReader fr;
        try{
              fr = new FileReader(algfile);
        }
        catch(FileNotFoundException e)
        {
              System.out.println("Shit!");
              return;
        }
```

```
            while(index != -1)
            {
                  try
                  {
                        index = fr.read(nextc, 0, 1);
                  }
                  catch(IOException e)
                  {
                        break;
                  }
                  if(nextc[0] == '\n')
                  {
                        algtext.addElement(currentline);
                        currentline = new String("");
                  }
                  else
                  {
                        if(index != -1)
                        {
                              currentline = currentline.concat((new
Character(nextc[0])).toString());
                        }
                  }
                  if(index == -1)
                  {
                        algtext.addElement(currentline);
                        currentline = new String("");
                  }
            }
            this.parsecode();
            Enumeration e = myTransitions.elements();
            while(e.hasMoreElements())
            {
                  Transition a = (Transition) e.nextElement();
                  System.out.println(a.getName());
            }
            return;
      }

      public Enumeration connections()
      {
            return myConnections.elements();
      }

      private void parsecode()
      {
            Enumeration code = algtext.elements();
            String tempstr;
            String vartype;
            int mode = 0;
            int trash_index = 0;
            boolean modeend = false;
            while(code.hasMoreElements())
            {
                  String line = (String)code.nextElement();
                  line = line.trim();
                  System.out.println(line);
                  if(line.equals("states"))
                  {
                        mode = 1;
                        System.out.println("In state mode.");
                        continue;
                  }
                  if(line.equals("transitions"))
```

```
                {
                        mode = 2;
                        System.out.println("In transition mode.");
                        continue;
                }
                if(mode == 1)
                {
                        trash_index = line.indexOf(':');
                        if(trash_index == -1)
                        {
                                parsefail();
                        }
                        else
                        {
                                String varname = line.substring(0,
trash_index);
                                varname = varname.trim();
                                tempstr = line.substring(trash_index+1);
                                trash_index = tempstr.indexOf(':');
                                if(trash_index < 0)
                                {
                                        vartype = tempstr.substring(1);
                                }
                                else
                                {
                                        vartype = tempstr.substring(1,
trash_index);
                                }
                                vartype = vartype.trim();
                                if(line.endsWith(","))
                                {
                                        mode = 1;
                                }
                                else
                                {
                                        mode = 0;
                                }
                                vartype = vartype.substring(0,
vartype.length() - 1);
                                vartype = vartype.trim();
                                System.out.println("Found a variable.");
                                this.addVariable(new Variable("", varname,
vartype, this.myRadius));
                        }
                }
                if(mode == 2)
                {
                        trash_index = line.indexOf(' ');
                        if(trash_index != -1)
                        {
                                String transtype = line.substring(0,
trash_index);
                                System.out.println("A transition?");
                                System.out.println(transtype);
                                transtype = transtype.trim();
                                String transname =
line.substring(trash_index + 1);
                                int numargs = 0;
                                if((transname.indexOf('(') + 1) ==
(transname.indexOf(')')))
                                {
                                        numargs = 0;
                                }
                                else
```

```
                                {
                                      numargs = 1;
                                      String temp = transname;
                                      int index = transname.indexOf(',');
                                      while(index >= 0)
                                      {
                                            numargs += 1;
                                            temp =
temp.substring(temp.indexOf(',') + 1);
                                            index = temp.indexOf(',');
                                      }
                                }
                                if((transtype.equals("input")) ||
                                   (transtype.equals("output")) ||
                                   (transtype.equals("internal")))
                                {
                                      Transition t2 = new
Transition(transname, transtype);
                                      t2.setNumArgs(numargs);
                                      this.addTransition(t2);
                                }
                          }
                    }
              }
              return;
        }

        private void parsefail()
        {
              System.out.println("Parse failed...badly.");
              return;
        }

        private boolean closeto(int x, int y)
        {
              if((x > (myRect.x)) && (x < (myRect.x + (myRect.width))))
              {
                    if ((y > (myRect.y)) && (y < (myRect.y +
(myRect.height))))
                    {
                          return true;
                    }
              }
              return false;
        }

        public boolean hitTest(int x, int y) {
              if ( this.isVisible() && this.closeto(x, y))
              {
                    System.out.println("Hit!");
                    return true;

              }
              return false;
        }

        public void setpos(int x, int y)
        {
              int diffx = x - myRect.x;
              int diffy = y - myRect.y;
              myRect.x = x;
              myRect.y = y;
              Enumeration enum = myState.elements();
              int minX = myRect.x + myFont.getSize();
```

```
            int minY = myRect.y + (2* myFont.getSize());
            int maxX = myRect.x + myRect.width;
            int maxY = myRect.y + myRect.height;
            minY += myFont.getSize();
            while (enum.hasMoreElements())
            {
                  Variable nextV = (Variable)enum.nextElement();
                  nextV.configure(minX, minY, maxX, maxY,
myFont.getSize());
                  minY += myFont.getSize();
            }
            enum = myTransitions.elements();
            minY += myFont.getSize();
            while(enum.hasMoreElements())
            {
                  Transition nextt = (Transition)enum.nextElement();
                  nextt.configure(minX, minY, maxX, maxY,
myFont.getSize());
                  minY += myFont.getSize();
            }
            enum = myConnections.elements();
            while(enum.hasMoreElements())
            {
                  Connection nextc = (Connection)enum.nextElement();
                  nextc.update();
            }
            if(!attachmoved)
            {
                  attachx = myRect.x;
                  attachy = myRect.y;
            }
            else
            {
                  attachx += diffx;
                  attachy += diffy;
            }


      }

      public void addVariable (Variable v)
      {
            myState.addElement((Object)v);
      }

      public void addTransition(Transition v)
      {
            myTransitions.addElement((Object)v);
      }

      public String getName() { return myName; }

      public void setName(String name) {
            myName = name;
            //repaint();
      }

      public Color getColor() { return myColor; }

      public void setColor(Color color) {
            myColor = color;
            //repaint();
      }
```

```
	public int getX() { return myRect.x; }

	public int getY() { return myRect.y; }

	public void setX(int x) {
		int diffx = x - myRect.x;
		myRect.x = x;
		if(attachmoved)
		{
			attachx += diffx;
		}
		else
		{
			attachx = x;
		}
		//repaint();
	}
	public void setY(int y) {
		int diffy = y - myRect.y;
		myRect.y = y;
		if(attachmoved)
		{
			attachy += diffy;
		}
		else
		{
			attachy = y;
		}
		//repaint();
	}
	public int getRadius() { return myRadius; }
	public int getWidth() { return myRect.width;}
	public int getHeight() { return myRect.height;}
	public boolean getshorthand() { return shorthand;}
	public File getFile() { return af;}
	public void setSize(int w, int h)
	{
		myRect.width = w;
		myRect.height = h;
		double dw = (double) ((((float)myRect.width)/2) *
(((float)myRect.width)/2));
		double dh = (double) ((((float)myRect.height)/2) *
(((float)myRect.height)/2));
		myRadius = (int) (Math.sqrt(dw + dh));
	}

	public String getFontName()
	{
		return myFont.toString();
	}

	public void setFont(String n)
	{
		myFont = new Font(n, Font.PLAIN, 12);
	}

	public void paint(Graphics g) {
		// only draw if this is visible
		if (isVisible())
		{
			// draw this ParticipationBean
			draw(g);

		}
```

```
	}

	public Transition getTransAt(int x, int y)
	{
		Enumeration enum = myTransitions.elements();
		int fontsize = myFont.getSize();
		int minx, miny, maxx, maxy;
		Transition t;
		while(enum.hasMoreElements())
		{
			t = (Transition)enum.nextElement();
			minx = t.getX();
			miny = t.getY() - fontsize;
			maxx = minx + (t.getLength() * fontsize);
			maxy = t.getY();
			if((x>=minx) && (x<=maxx) && (y>=miny) && (y<=maxy))
			{
				return t;
			}
		}
		return (new Transition("SWMError", "Error"));
	}

	public void addConnection(Connection c)
	{
		myConnections.addElement(c);
	}

	public boolean connect(ProcessBean pb, int x, int y)
	{
		Transition t1 = pb.getTransAt(x, y);
		Connection c = new Connection(this, pb);
		boolean exists = false;
		if((t1.getType()).equals("Error"))
		{
			return false;
		}
		System.out.println("Made it past transition finder.");
		Transition t2;
		Enumeration enum = myTransitions.elements();
		while(enum.hasMoreElements())
		{
			t2 = (Transition)enum.nextElement();
			System.out.println("-----------");
			System.out.println(t2.getProperName());
			System.out.println(t2.getType());
			System.out.println(t1.getProperName());
			System.out.println(t1.getType());
			if((t2.getProperName().equals(t1.getProperName())) &&
			   (t2.getType().equals("output")) &&
			   (t1.getType().equals("input")))
			   {
			      System.out.println("Found a connection to
make.");
			      Enumeration enum2 = myConnections.elements();
			      while(enum2.hasMoreElements() && !exists)
			      {
					c = (Connection)enum2.nextElement();
					if(c.exists(this, pb))
					{
						exists = true;
						System.out.println("Exists
already?");
						break;
```

```
                        }
                    }
                    if(!exists)
                    {
                        c = new Connection(this, pb);
                        Enumeration enum3 = pb.connections();
                        while(enum3.hasMoreElements())
                        {
                            Connection c2 =
(Connection)enum3.nextElement();
                            if(c2.exists(pb, this))
                            {
                                c.setType(2);
                                break;
                            }
                        }
                    }
                    String newname;
                    if(t1.getNumArgs() > t2.getNumArgs())
                    {
                        newname = t1.getName() + "-" +
this.getName() + "-" + pb.getName();
                    }
                    else
                    {
                        newname = t2.getName() + "-" +
this.getName() + "-" + pb.getName();
                    }
                    c.addComp(t2, t1, newname);
                    if(!exists)
                    {
                        c.setColor1(color1);
                        c.setColor2(color2);
                        myConnections.addElement(c);
                    }
                    System.out.println("Success!");
                    return true;
                }
            }
            return false;
        }

        public int getAttachx()
        {
            return attachx;
        }

        public int getAttachy()
        {
            return attachy;
        }

        public boolean connect2(ProcessBean pb, Transition t1)
        {
            Connection c = new Connection(this, pb);
            boolean exists = false;
            if((t1.getType()).equals("Error"))
            {
                return false;
            }
            System.out.println("Made it past transition finder.");
            Transition t2;
            Enumeration enum = myTransitions.elements();
            while(enum.hasMoreElements())
```

```
            {
                  t2 = (Transition)enum.nextElement();
                  System.out.println("-----------");
                  System.out.println(t2.getProperName());
                  System.out.println(t2.getType());
                  System.out.println(t1.getProperName());
                  System.out.println(t1.getType());
                  if((t2.getProperName().equals(t1.getProperName())) &&
                     (t2.getType().equals("output")) &&
                     (t1.getType().equals("input")))
                     {
                        System.out.println("Found a connection to
make.");
                        Enumeration enum2 = myConnections.elements();
                        while(enum2.hasMoreElements() && !exists)
                        {
                              c = (Connection)enum2.nextElement();
                              if(c.exists(this, pb))
                              {
                                    exists = true;
                                    System.out.println("Exists
already?");
                                    break;
                              }
                        }
                        if(!exists)
                        {
                              c = new Connection(this, pb);
                              Enumeration enum3 = pb.connections();
                              while(enum3.hasMoreElements())
                              {
                                    Connection c2 =
(Connection)enum3.nextElement();
                                    if(c2.exists(pb, this))
                                    {
                                          c.setType(2);
                                          break;
                                    }
                              }
                        }
                        String newname = t2.getName() + "-" +
this.getName() + "-" + pb.getName();
                        c.addComp(t2, t1, newname);
                        if(!exists)
                        {
                              c.setColor1(color1);
                              c.setColor2(color2);
                              myConnections.addElement(c);
                        }
                        System.out.println("Success!");
                        return true;
                  }
            }
            return false;
      }


      public void SetVarValue(String variable, String value)
      {
            Enumeration enum = myState.elements();
            while(enum.hasMoreElements())
            {
                  Variable v = (Variable)enum.nextElement();
                  String oldval = v.getValue();
```

```
                if(v.getName().equals(variable))
                {
                        v.setValue(value);
                        if(!value.equals(oldval))
                        {
                                v.setUsed();
                        }
                        break;
                }
        }
    }

    public void SetTransValue(String dest, String trans, String value)
    {
        Enumeration enum = myConnections.elements();
        while(enum.hasMoreElements())
        {
                Connection c = (Connection)enum.nextElement();
                if(c.getDestination().getName().equals(dest))
                {
                        Enumeration enum2 = c.getCompositions();
                        while(enum2.hasMoreElements())
                        {
                                Composition m =
(Composition)enum2.nextElement();
                                String s = m.getSource().getName();
                                s = s.substring(0, s.indexOf('('));
                                if(s.equals(trans))
                                {
                                        m.getSource().setArgs(value);
                                        m.getSource().setUsed();
                                        m.getDestination().setUsed();
                                        m.getDestination().setArgs(value);
                                        m.setArgs(value);
                                        m.setUsed();
                                        c.setUsed();
                                }
                        }
                }
        }
    }


    public void reset()
    {
        Enumeration enum = myState.elements();
        while(enum.hasMoreElements())
        {
                Variable v = (Variable)enum.nextElement();
                v.unsetUsed();
        }
        enum = myTransitions.elements();
        while(enum.hasMoreElements())
        {
                Transition t = (Transition)enum.nextElement();
                t.unsetUsed();
                t.reset();
        }
        enum = myConnections.elements();
        while(enum.hasMoreElements())
        {
                Connection c = (Connection)enum.nextElement();
                c.unsetUsed();
        }
```

```
	}

	public boolean inSpot(int x, int y)
	{
		if(!shorthand)
		{
			return false;
		}
		if((x > hotspot.x) && (x < (hotspot.x + hotspot.width)) &&
(y > hotspot.y) && (y < (hotspot.y + hotspot.height)))
		{
			return true;
		}
		return false;
	}

	public void setAttach(int x, int y)
	{
		attachmoved = true;
		if((x < myRect.x) && (y < myRect.y))
		{
			attachx = myRect.x;
			attachy = myRect.y;
			return;
		}
		if((x < myRect.x) && (y  > (myRect.y + myRect.height)))
		{
			attachx = myRect.x;
			attachy = myRect.y + myRect.height;
			return;
		}
		if((x > (myRect.x + myRect.width)) && (y < myRect.y))
		{
			attachx = myRect.x + myRect.width;
			attachy = myRect.y;
			return;
		}
		if((x > (myRect.x + myRect.width)) && (y > (myRect.y +
myRect.height)))
		{
			attachx = myRect.x + myRect.width;
			attachy = myRect.y + myRect.height;
			return;
		}
		if((y < myRect.y))
		{
			attachx = x;
			attachy = myRect.y;
			return;
		}
		if(y > (myRect.y + myRect.height))
		{
			attachx = x;
			attachy = myRect.y + myRect.height;
			return;
		}
		if(x < myRect.x)
		{
			attachx = myRect.x;
			attachy = y;
			return;
		}
		if(x > (myRect.x + myRect.width))
		{
```

```
                attachx = myRect.x + myRect.width;
                attachy = y;
                return;
            }
            attachx = myRect.x;
            attachy = myRect.y;
            return;
        }


        public Enumeration getVariables()
        {
            return myState.elements();
        }

        public Enumeration getTransitions()
        {
            return myTransitions.elements();
        }

        public Enumeration getConnections()
        {
            return myConnections.elements();
        }

        public boolean isShorthand()
        {
            return shorthand;
        }

        public void setFontSize(int newsize)
        {
            myFont = new Font(myFont.getName(), myFont.getStyle(),
newsize);
        }

        public void setColor1(Color a)
        {
            Enumeration enum = myConnections.elements();
            while(enum.hasMoreElements())
            {
                Connection c = (Connection)enum.nextElement();
                c.setColor1(a);
            }
            color1 = a;
        }

        public void setColor2(Color a)
        {
            Enumeration enum = myConnections.elements();
            while(enum.hasMoreElements())
            {
                Connection c = (Connection)enum.nextElement();
                c.setColor2(a);
            }
            color2 = a;
        }


        private void draw(Graphics g) {
            // draw label on top left of roundRect
            shorthand = false;
            int Yoffset = 0;
```

```
            int Xoffset = myRadius/2;
            g.setFont(myFont);
            FontMetrics fm = g.getFontMetrics(myFont);
            if(current)
            {
                  g.setColor(Color.red);
            }
            else
            {
                  g.setColor(Color.black);
            }
            g.drawString(myName, myRect.x + 2, myRect.y);

            // draw rounded rectangle with border
            g.setColor(myColor);
            g.fillOval((myRect.x + 1), (myRect.y + 1), (myRect.width -
2), (myRect.height - 2));
            g.fillOval((attachx - 1), (attachy - 1), 2, 2);
            if(current)
            {
                  g.setColor(Color.red);
            }
            else
            {
                  g.setColor(Color.black);
            }
            g.drawOval((myRect.x + 1), (myRect.y + 1), (myRect.width -
2), (myRect.height- 2));
            Enumeration enum = myState.elements();
            int minX = myRect.x + myFont.getSize();
            int minY = myRect.y + (2* myFont.getSize());
            int maxX = myRect.x + myRect.width;
            int maxY = myRect.y + myRect.height;
            g.drawString("State:", minX, minY);
            minY += myFont.getSize();
            while (enum.hasMoreElements())
            {
                  Variable nextV = (Variable)enum.nextElement();
                  nextV.configure(minX, minY, maxX, maxY,
myFont.getSize());
                  minY += myFont.getSize();
                  if(minY > maxY)
                  {
                        if(!shorthand)
                        {
                              shorthand = true;
                              g.drawString("...", minX, minY);
                        }
                  }
                  if(!shorthand)
                  {
                        nextV.paint(g);
                  }
            }
            if(current)
            {
                  g.setColor(Color.red);
            }
            else
            {
                  g.setColor(Color.black);
            }
            g.setFont(myFont);
            enum = myTransitions.elements();
```

```
            g.drawString("Transitions", minX, minY);
            minY += myFont.getSize();
            while(enum.hasMoreElements())
            {
                  Transition nextt = (Transition)enum.nextElement();
                  nextt.configure(minX, minY, maxX, maxY,
myFont.getSize());
                  minY += myFont.getSize();
                  if(minY > maxY)
                  {
                        if(!shorthand)
                        {
                              shorthand = true;
                              g.drawString("...", minX, minY);
                              hotspot = new Rectangle(minX, minY -
myFont.getSize(), 4, myFont.getSize());
                        }
                  }
                  if(!shorthand)
                  {
                        nextt.paint(g);
                  }
            }
            enum = myConnections.elements();
            while(enum.hasMoreElements())
            {
                  Connection nextc = (Connection)enum.nextElement();
                  nextc.paint(g);
            }
      }

      public Dimension getPreferredSize() {
            return new Dimension(2 * myRadius, 2 * myRadius);
      }


}
```

Variable.java

```
import java.util.*;
import java.awt.*;
public class Variable
{
      private String myValue;
      private String myType;
      private String myName;
      private Vector myValues;
      private int firstindex = 0;
      private int lastindex = 0;
      private int myX = 0;
      private int myY = 0;
      private Color myColor;
      private boolean myVis = true;
      private boolean changed = false;
      private boolean shorthand = false;
      private int availableRadius = 0;
      private Rectangle hotspot;

      public Variable(String v, String n, String t, int a)
      {
            myValue = v;
            myName = n;
            myType = t;
            availableRadius = a;
```

```
}

public void setUsed()
{
      changed = true;
}

public void unsetUsed()
{
      changed = false;
}

public boolean isVisible()
{
      return myVis;
}

public String getName()
{
      return myName;
}

public String getType()
{
      return myType;
}

public String getValue()
{
      return myValue;
}

public int getLength()
{
      return (myName + ":" + myValue).length();
}

public void setValue(String newval)
{
      myValue = newval;
      return;
}

public boolean isShorthand()
{
      return shorthand;
}

public boolean hittest(int x, int y, int fsize)
{
      int myLength = this.getLength() * fsize;
      int myHeight = fsize;
      if((x > myX) && (x < (myX + myLength)) &&
         (y > myY - myHeight) && (y < (myY)))
      {
            return true;
      }
      else
      {
            return false;
      }
}
```

```
	public void configure(int minx, int miny, int maxx, int maxy, int
fsize)
	{
		this.setPos(minx, miny);
		shorthand = false;
		if((this.getLength() * fsize) > maxx)
		{
			shorthand = true;
			int p = myName.length() * fsize;
			hotspot = new Rectangle(myX + p, myY - fsize, 4,
fsize);
		}
		else
		{
			shorthand = false;
		}
	}

	public boolean inSpot(int x, int y)
	{
		if(!shorthand)
		{
			return false;
		}
		if((x > hotspot.x) && (x < (hotspot.x + hotspot.width)) &&
(y > hotspot.y - hotspot.height) && (y < (hotspot.y)))
		{
			return true;
		}
		return false;
	}

	public void setPos(int x, int y)
	{
		myX = x;
		myY = y;
	}

	public void paint(Graphics g) {
		// only draw if this is visible
		if (isVisible())
		{
			// draw this ParticipationBean
			draw(g);

		}
	}


	private void draw(Graphics g) {
		// draw label on top left of roundRect
		if(changed)
		{
			g.setColor(Color.red);
		}
		else
		{
			g.setColor(Color.black);
		}
		String myString = myName + ":" + myValue;
		if(!shorthand)
		{
			g.drawString(myString, myX, myY);
		}
```

```
            else
            {
                  myString = myName + ":...";
                  g.drawString(myString, myX, myY);
            }

      }

}
```

Transition.java

```
import java.util.*;
import java.awt.*;
public class Transition
{
      private String myName;
      private String myProperName;
      private String myBaseName;
      private String myType;
      private int myX = 0;
      private int myY = 0;
      private Color myColor;
      private boolean used = false;
      private boolean myVis = true;
      private int numargs;


      public Transition(String n, String t)
      {
            myName = n;
            myProperName = myName.substring(0, myName.indexOf('('));
            myType = t;
            myBaseName = myName;
      }

      public void setArgs(String args)
      {
            int i = 1;
            String g = args;
            String temp2 = "";
            while(i<=numargs)
            {
                  int q = g.indexOf(',');
                  String val;
                  if(q < 0)
                  {
                        val = g;
                        temp2 = temp2 + val;
                        break;
                  }
                  else
                  {
                        val = g.substring(0, g.indexOf(','));
                        temp2 = temp2 + val + ", ";
                        g = g.substring(g.indexOf(',') + 1);
                  }
            }
            String temp = myName.substring(0, myName.indexOf('(')+1);
            myName = temp + temp2 + ")";
      }

      public void setUsed()
      {
            used = true;
```

```
	}

	public void setNumArgs(int i)
	{
		numargs = i;
	}

	public int getNumArgs()
	{
		return numargs;
	}

	public void unsetUsed()
	{
		used = false;
	}

	public boolean isVisible()
	{
		return myVis;
	}

	public String getName()
	{
		return myName;
	}

	public String getProperName()
	{
		return myProperName;
	}

	public String getType()
	{
		return myType;
	}

	public int getLength()
	{
		return myName.length();
	}

	public void configure(int minx, int miny, int maxx, int maxy, int
fsize)
	{
		this.setPos(minx, miny);
	}

	public void setPos(int x, int y)
	{
		myX = x;
		myY = y;
	}

	public int getX()
	{
		return myX;
	}

	public int getY()
	{
		return myY;
	}
```

```
	public void reset()
	{
		myName = myBaseName;
	}

	public void paint(Graphics g) {
		// only draw if this is visible
		if (isVisible())
		{
			// draw this ParticipationBean
			draw(g);

		}
	}

	private void draw(Graphics g) {
		// draw label on top left of roundRect
		if(used)
		{
			g.setColor(Color.red);
		}
		else
		{
			g.setColor(Color.black);
		}
		String myString;
		myString = myName;
		g.drawString(myString, myX, myY);
	}

}
```

Connection.java

```
import java.util.*;
import java.awt.*;
import Transition.*;
import Composition.*;
import ProcessBean.*;

public class Connection
{
	private ProcessBean source;
	private ProcessBean destination;
	private int sourceX = 0;
	private int sourceY = 0;
	private int destX = 0;
	private int destY = 0;
	private Vector compositions;
	private boolean myVis = true;
	private boolean used = false;
	private int myType = 1;
	private Color color1 = Color.blue;
	private Color color2 = Color.green;

	public Connection(ProcessBean s, ProcessBean d)
	{
		source = s;
		destination = d;
		sourceX = source.getAttachx();
		sourceY = source.getAttachy();
		destX = destination.getAttachx();
		destY = destination.getAttachy();
		compositions = new Vector();
	}
```

```
	public void setType(int i)
	{
		myType = i;
		if(myType != 1)
		{
			sourceX -= 5;
			sourceY -= 5;
		}
	}

	public boolean exists(ProcessBean s, ProcessBean d)
	{
		if(source.equals(s) && destination.equals(d))
		{
			return true;
		}
		else
		{
			return false;
		}
	}

	public void addComp(Transition s, Transition d, String name)
	{
		boolean done = false;
		Composition c;
		Enumeration enum = compositions.elements();
		while(enum.hasMoreElements())
		{
			c = (Composition)enum.nextElement();
			if((c.getSource().equals(s)) &&
(c.getDestination().equals(d)))
			{
				done = true;
			}
		}
		if(!done)
		{
			compositions.addElement(new Composition(s, d, name));
		}
		return;
	}

	public boolean isVisible()
	{
		return myVis;
	}

	public ProcessBean getSource()
	{
		return source;
	}

	public ProcessBean getDestination()
	{
		return destination;
	}

	public void update()
	{
		sourceX = source.getAttachx();
		sourceY = source.getAttachy();
		destX = destination.getAttachx();
```

```
            destY = destination.getAttachy();
            if(myType != 1)
            {
                  sourceY -= 5;
                  destY -= 5;
            }
            return;
      }

      public void setUsed()
      {
            used = true;
      }

      public void unsetUsed()
      {
            used = false;
            Enumeration enum = compositions.elements();
            while(enum.hasMoreElements())
            {
                  Composition c = (Composition)enum.nextElement();
                  c.unsetused();
            }
      }

      public Enumeration getCompositions()
      {
            return compositions.elements();
      }

      public void paint(Graphics g) {
            // only draw if this is visible
            if (isVisible())
            {
                  // draw this ParticipationBean
                  draw(g);

            }
      }

      public void setColor1(Color a)
      {
            color1 = a;
      }

      public void setColor2(Color b)
      {
            color2 = b;
      }

      private void draw_arrow(int pointx, int pointy, int basex, int
basey, Graphics g)
      {
            int diffx = pointx - basex;
            int diffy = pointy - basey;
            int point1x = diffx - 10;
            int point1y = diffy + 5;
            int point2y = diffy - 5;
            g.drawLine(pointx, pointy, basex + point1x, basey +
point1y);
            g.drawLine(pointx, pointy, basex + point1x, basey +
point2y);
      }
```

```
	private void draw(Graphics g) {
		// draw label on top left of roundRect
		if(used)
		{
			g.setColor(Color.red);
		}
		else
		{
			if(myType == 1)
			{
				g.setColor(color1);
			}
			else
			{
				g.setColor(color2);
			}

		}
		g.drawLine(sourceX, sourceY, destX, destY);
		double vectX = (double)sourceX - (double)destX;
		double vectY = (double)sourceY - (double)destY;
		double mag = Math.sqrt((vectX * vectX) + (vectY * vectY));
		vectX = vectX/mag;
		vectY = vectY/mag;
		vectX = vectX * 10;
		vectY = vectY * 10;
		//draw_arrow(destX, destY, destX + (int)vectX, destY +
(int)vectY, g);
		g.fillOval(destX, destY, 4, 4);
		int midpointX;
		int midpointY;
		if(sourceX<destX)
		{
			midpointX = sourceX + ((destX - sourceX)/2);
		}
		else
		{
			midpointX = destX + ((sourceX - destX)/2);
		}
		if(sourceY<destY)
		{
			midpointY = sourceY + ((destY - sourceY)/2);
		}
		else
		{
			midpointY = destY + ((sourceY - destY)/2);
		}
		if(sourceX < destX)
		{
			midpointY = midpointY + (g.getFont()).getSize();
		}
		else
		{
			midpointY = midpointY - (g.getFont()).getSize();
		}
		Enumeration enum = compositions.elements();
		while(enum.hasMoreElements())
		{
			Composition c = (Composition)enum.nextElement();
			if(c.isused())
			{
				g.setColor(Color.red);
			}
			else
```

```
                {
                        if(myType == 1)
                        {
                                g.setColor(Color.blue);
                        }
                        else
                        {
                                g.setColor(Color.green);
                        }
                }
                g.drawString(c.getName(), midpointX, midpointY);
                if(sourceX < destX)
                {
                        midpointY = midpointY + (g.getFont()).getSize();
                }
                else
                {
                        midpointY = midpointY - (g.getFont()).getSize();
                }
        }
    }

}
```

Composition.java

```
import java.util.*;
import java.awt.*;
import Transition.*;

public class Composition
{
    private Transition source;
    private Transition destination;
    private String name;
    private String basename;
    private boolean used = false;

    public Composition(Transition s, Transition d, String n)
    {
        source = s;
        destination = d;
        name = n;
        basename = name;
    }

    public Transition getSource()
    {
        return source;
    }

    public Transition getDestination()
    {
        return destination;
    }

    public String getName()
    {
        return name;
    }

    public void setArgs(String args)
    {
        String first = name.substring(0, name.indexOf('(') + 1);
```

```
            String rest = name.substring(name.indexOf(')'));
            first = first + args + rest;
            name = first;
      }

      public boolean isused()
      {
            return used;
      }

      public void setUsed()
      {
            used = true;
      }

      public void unsetused()
      {
            used = false;
            name = basename;
      }
}
```

488-37