SAOSDB:
A Structural Active-Object System Debugger

Pornsiri Muenchaisri

Dept. of Computer Science
Oregon State University

Structural Active-Object System Debugger

Pornsiri Muenchaisri
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-4602
muenchp@research.cs.orst.edu

Abstract

Structural active-object systems (SAOSs) are transition-based object-oriented systems suitable for various concurrent applications. A SAOS consists of a collection of interacting active objects that can be *structurally* and *hierarchically* composed. The SAOS framework enables SAOS application programs to be developed rapidly. However, nondeterministic behaviors of active objects sometimes make debugging SAOS application programs difficult. To overcome this problem, Structural Acitve-Object System Debugger (SAOSDB) was designed and implemented as an *object-based* visual debugging tool. SAOSDB allows a programmer to see directly the states of the active objects and to control interactively the executions of the behavior specification statements of active objects. SAOSDB itself is implemented as a SAOS.

Key Words and Phrases: object-oriented programming, object-based debugger, visual debugger, active-object system, structural composition, hierarchical composition

Contents

1 Introduction

A *structural active-object system* (SAOS) is an *object-oriented concurrent* system that is *structurally* and *hierarchically* constructed from a collection of active objects. The behaviors of the active objects are defined by *transition statements*, which are *transition rules*, `always` *statements*, *future calls*, or *future assignments* [MINO93b]. A unique feature of the SAOS approach is that we can obtain a fully functional program by specifying only the structural relationships among its components.

For a new SAOS programmer, however, debugging a complex SAOS application program is not trivial since it is organized in a different way from conventional programs. The programmer must understand what active objects are created, how they interact with each other, and how transition rules, event routines, and equational assignment statements work. Especially, nondeterministic executions of transition statements make debugging a SAOS program difficult and time-consuming. In order to ease the burden of debugging SAOS application programs, Structural Active Object System Debugger (SAOSDB) was developed. SAOSDB is a debugger that maximizes the benefits of the SAOS approach.

Traditional debuggers are *statement-based*, with their focus on sequentially executable statements. The executions of sequential statements can be suspended by breakpoints, or they can be performed one at a time by single-stepping. Statement-based debuggers have the following problems.

- A statement-based debugger does not provide an overall picture of the program being debugged, thus it is not easy to know what objects are created and how they are interrelated with each other.
- A traditional debugger does not provide an easy way for a programmer to retrieve information for a particular object and switch to another object.
- It is impossible to set a breakpoint for a method of a particular object.

Purchase states that an object-oriented software debugger should support debugging at the object level [PURS91]. The SAOSDB, an *object-based* debugger, was designed and developed to support object-based debugging of SAOS programs.

By using SAOSDB, a programmer can display the SAOS program being debugged as a hierarchy of active objects. Displaying an overall picture of the program being debugged

makes comprehension of the overall structure of the program easier. The state of each active object can be inspected during debugging. Activated transitions can be executed one at a time by single-stepping. Furthermore, it allows the user to view the sequence of actions that occur to any particular object.

Section 2 introduces the SAOS approach. Section 3 describes the Active-Object User Interface Management System (AOUIMS), whose functionalities are utilized by SAOSDB. The major features of SAOSDB are summarized in Section 4. Section 5 discusses the implementation details of SAOSDB. Section 6 concludes this report.

2 Overview of SAOSDB

The major objective of SAOSDB is to provide an *object-based* debugging environment for SAOS programs, exploiting the unique features of the SAOS approach.

1. In order to help the programmer to comprehend the overall structure of a SAOS program being debugged, the active objects created by the program are displayed as a hierarchical tree structure as organized in the SAOS program.

2. An execution of the program being debugged can be suspended at any transition statement boundary, and the states of any selected active objects can be printed.

3. The transition statements can be executed one at a time (single-stepping). Single-stepping can be performed for the whole program or for a particular active object selected by the user.

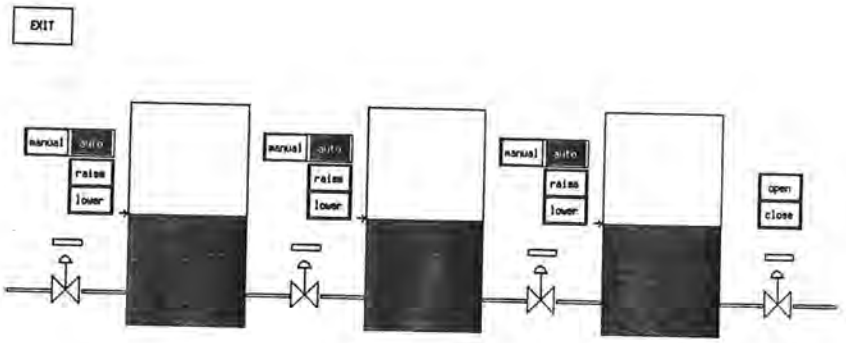

Figure 1: Graphical Interface of the Tank-System Simulator.

Fig. 1 shows the graphical user interface of the tank-system simulator, which is a SAOS application program. This program is described in [MINO93c].

Fig. 2 shows the graphical user interface of SAOSDB while the tank-system simulator is being debugged. Each active object is represented by a rectangle with the name of the active object in it. The `TopLevel` active object is named `3tankSys`, which is the whole tank-system simulator program, and its descendants are displayed on the canvas as a tree structure. The

component active objects of `3tankSys` are `vTank0`, `vTank1`, `vTank2`, `lastManBtn`, `lastValve`, `manControl`, and `exitButton`.

Each component may consist of subcomponets. For example, the component `vTank2`, which is the third valve-tank-controller subsystem in Fig. 1, has two subcomponents `valveSubsys` and `tank`. The `valveSubsys` has six subcomponents `manAuto`, `manBtn`, `autoBtn`, `refArrow`, `vPipe`, and `manControl`. The `vPipe` has 4 subcomponents `opening`, `valveElem`, `pipeA`, and `pipeB`, and so on.

The programmer can interact with SAOSDB by pressing the buttons provided above the canvas. The buttons `Model` and `View` are used to select the active objects to be displayed. The active objects must be either in `model` or `view`. The second group of buttons (`Run`, `GPause`, `GNext`, ..., and `Print`) are used to control an execution of the program being debugged and to inspect the active objects created by the program being debugged.

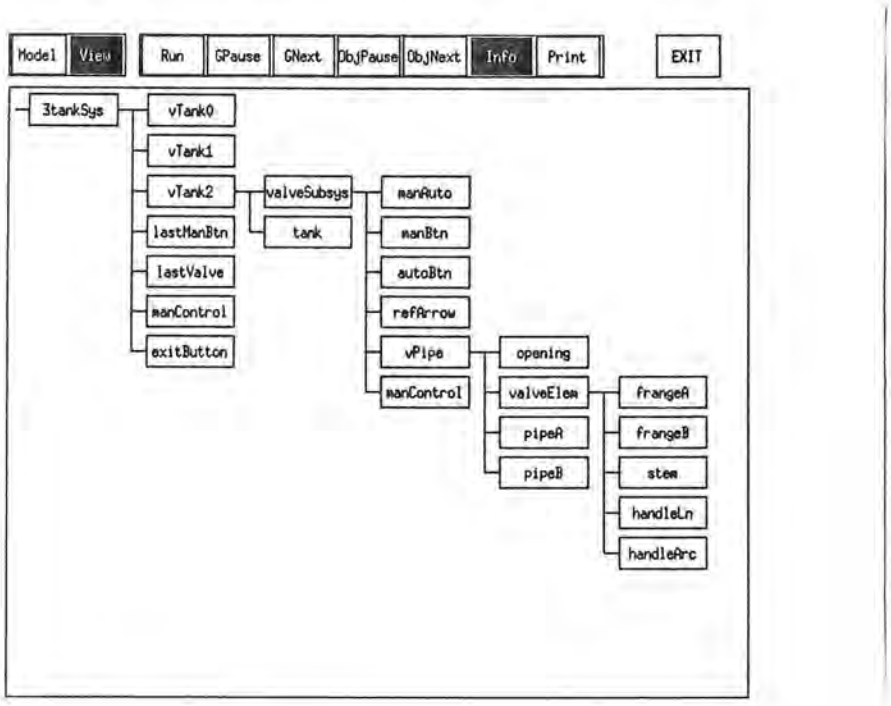

Figure 2: SAOSDB Graphical User Interface for the Tank-System Simulator.

2.1 User Operations

A SAOS application program can be run in the debugging mode, if the debugging option `-d` is provided in the command line as `tank3 -d`. The application program need not be modified.

SAOSDB supports the following debugging operations.

1. The SAOSDB allows the active objects either in `model` or in `view` to be displayed, while the `Model` (`View`) button is selected, objects in `model` (`view`) are displayed. When the program contains only one of `model` or `view`, whichever exists is selected. When the program contains both `model` and `view`, `model` is selected by default.

2. When the `Tree` button is pressed, SAOSDB enters the *tree-structure-display* mode. In this mode, when an object node on the canvas is selected by a mouse click, the nodes for its children objects are displayed. This process can be continued to display all the objects of a particular lineage. When an object node is deselected, the subtrees of its descendant nodes are removed from the canvas.

3. When the `GPause` (global pause) button is pressed, the executions of all user transition statements are suspended.

4. When the `GNext` (global next) button is pressed, while executions of user transition statements are suspended, one activated transition statement is executed as single-stepping.

5. When the `ObjPause` (object pause) button is pressed, SAOSDB enters the *object-pause* mode. In this mode, when an object node on the canvas is selected by a mouse click, the executions of the transition statements applicable to the selected object are suspended. Executions of transition statements for other objects are not suspended.

6. When the `ObjNext` (object next) button is pressed, one activated transition statement T_i of the selected object O, together with all the activated transition statements enqueued after T_i and before another activated transition statement T_{i+1} of O, is executed.

7. When the `Print` button is pressed, SAOSDB enters the *print* mode. In this mode, when an object node on the canvas is selected by a mouse click, the data of the selected object are printed.

Fig. 3 shows the graphical interface of the Queueing System simulator. Fig. 4 shows the

SAOSDB with the Queueing System Simulation being debugged in the `Model` mode, and Fig. 5 in the `View` mode.

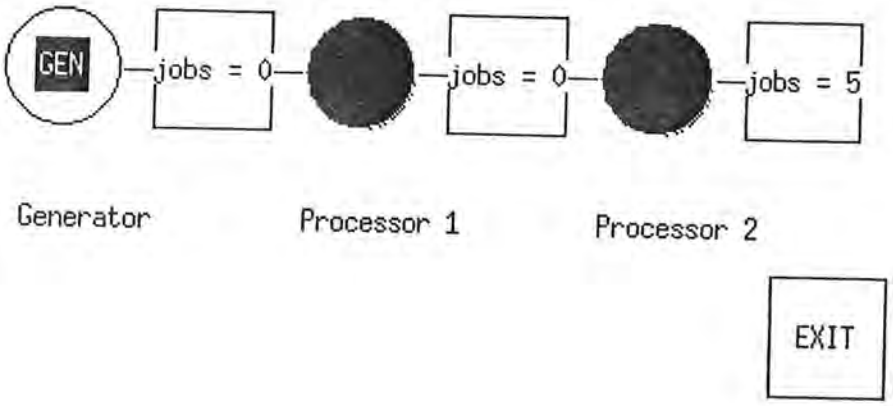

Figure 3: Graphical Interface of the Queueing System Simulator.

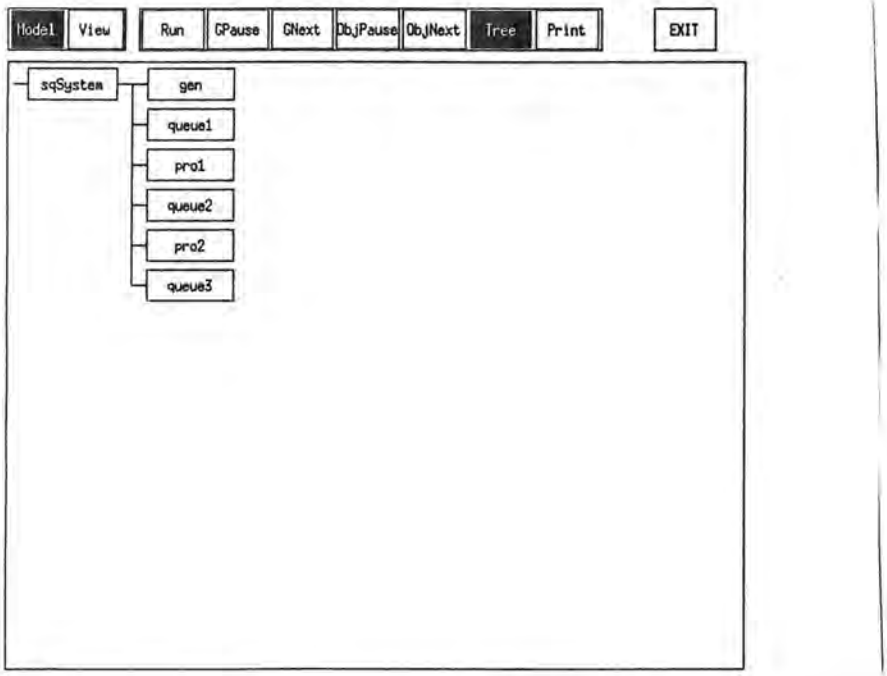

Figure 4: SAOSDB Graphical Interface for the `Model` of the Queueing System Simulator.

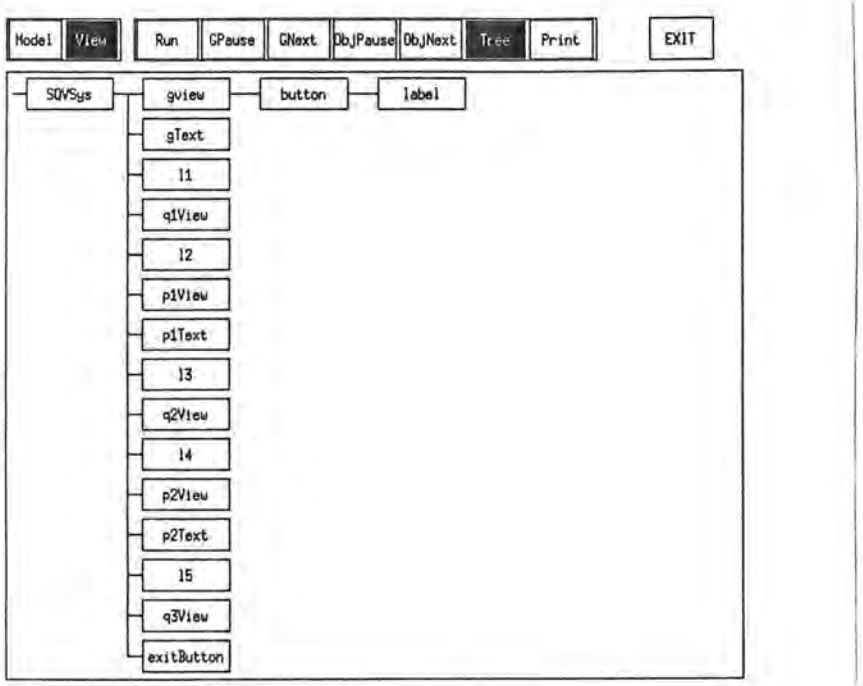

Figure 5: SAOSDB Graphical Interface for the `View` of the Queueing System Simulator.

2.2 SAOSDB Configuration

The SAOS application program with `SAOSDB` consists of the following components:

1. Active Object System (AOS) kernel,
2. Active-Object User-Interface Management System (AOUIMS),
3. SAOSDB code, and
4. User SAOS application code.

Developing a SAOS application program using the SAOS framework is not too complicated. Since the AOS kernel, AOUIMS, and SAOSDB code are embedded in the framework, a programmer needs to provide only user application code. The programmer does not need to modify the application code to activate the debugger, but to allow the user data of an object instantiated from a user-defined class to be inspected during debugging, it is necessary for the programmer to provide the `print()` method for the user-defined class.

3 The SAOS approach

Structural active-object systems (SAOSs) are constructed as a collection of structurally and hierarchically composed active objects. The behaviors of these active objects are defined by *transition statement*, which are *transition rules*, `always` *statements*, `future calls`, or `future assignments`. The active objects in a SAOS program are organized in a tree structure similar to that usd by the Unix file system.

In this section, we first describe the mechanisms for active behavior specifications for SAOs, and we then explain a component hierarchy of SAOS.

3.1 Active Behavior Description

Objects in Smalltalk or C++ are passive in the sense that they only respond to the messages sent to them. Besides *event routines* that respond to messages, a SAOS uses *transition rules*, which are *condition-action* pairs, `always` *statements*, which are *equational assignment statements*, and *event routines* for behavior descriptions of active objects. We refer to transition rules, `always` statements, and event routines as *transition statements*.

Each transition rule is a condition-action pair, whose action part is executed when its condition part is satisfied. An execution of a transition rule should be atomic. A transition rule is activated whenever the value of any *condition variable*, which is often called an *active value*, used in its condition part changes.

A simple mechanism for describing a behavior of an object is an equational assignment statement that maintains an invariant relationship (constraint) among the states of objects. Our `always` statements are used for this purpose. An `always` statement can be implemented like a transition rule. The execution of the `always` statement should be triggered whenever any of the variables used in the expression of the `always` statement changes. Triggered `always` statements are executed immediately at the points where they are activated, their executions being embedded within those of the transition statements.

The activations of transition rules and `always` statements are, at least conceptually, state-driven, and SAOs can communicate with each other by directly accessing the states of other objects through their interface variables rather than by sending messages to them. Although this mechanism mostly eliminates the necessity of activating functions (or methods) by explicit

events (or messages), some actions can be more efficiently handled by *event routines*, which are activated by *synchronous function calls*, *future calls*, or *future assignments*. Activated future calls and future assignments are executed one at a time like triggered transition rules. Synchronous function calls are supported by SAOS, but SAOSDB does not utilize this feature.

3.2 Component Hierarchy

A SAOS program maintains the component hierarchy on its component `AObjects`, which are usually constructed by structural and hierarchical object composition. If an `AObject` x is inserted into another `AObject` y, x becomes a child of y in the component hierarchy. Pointers `children`, `sibling`, and `parent` provided in `AObjects` are used to define the component hierarchy. If the `names` assigned to the `AObjects` are unique among the children of their respective parents, the function `pathName()` returns the unique *path name* of each `AObject`. This mechanism is similar to the file-naming mechanism of Unix.

After writing many SAOS programs in various application areas, we obtained the following insights on the relative usefulness among the behavior specification methods listed above. In most cases, control logic of simulation can be best described by transition rules. `always` statements are useful to update the attributes of graphical objects automatically. Inter-object synchronous function calls are rarely needed by application SAOS programs. Future calls and assignments are convenient to schedule delayed actions, although they are used less frequently compared to transition rules or `always` statements. On the other hand, such SAOS programs as AOUIMS and SAOS graphical editors use inter-object synchronous function calls in about 50% of their behavior specifications, and hence they look more similar to conventional OO programs.

4 The Active-Object User Interface Management System

The Active-Object User Interface Management System(AOUIMS) is a graphical user-interface management system (GUIMS) for *structural active-object systems* (SAOSs). Many application programs with dynamic (animated) graphical user interfaces have been created as SAOS programs. *Active user-interface objects* (AUIOs) supported by AOUIMS can be *structurally* and *hierarchically* composed from their component AUIOs, allowing graphical user interfaces to be constructed quickly. The implementation of AOUIMS itself follows the SAOS approach. In this section, we describe the principles and the structure of the *Active-Object User-Interface Management System* (AOUIMS). AOUIMS has the following unique features.

1. Behaviors of *active user-interface objects* (AUIOs) are mostly specified by transition rules and `always` statements, which allow concise descriptions and encapsulation of control.

2. A user interface is constructed by *structural* and *hierarchical* composition of AUIOs. Any AUIO can be a component of another AUIO. Thus, there is no special class for composite graphical objects.

3. The location and appearance of an AUIO is mostly specified by its *control variables*, whereas the results of user interactions are stored in *state variables*. Control and state variables can often replace complex procedural interfaces.

4. The above features in combination promote a *declarative* style of programming, and hence simplify the design and implementation of a graphical editor used to construct user-interfaces by *pick-and-place* operations.

5. Each AUIO can integrate the *model*, *view* and *control*. In this case, a graphical user-interface can be obtained with minimal extra effort.

6. It is possible to separate the model from the view and control. In this case, the same binary code of the model can be shared by the SAOS program with a graphical user-interface and the one without it.

Fig. 6 shows a part of the AOUIMS class hierarchy. Every `AObject` has a printable `name` and can be organized into a *component hierarchy*. A `Monitor`[1] is needed for each X server. It accepts X-events from the server and distributes them to appropriate `TopLevels` and `Popups`,

[1]From now on, when x is an instance of class X or one of its subclass, we simply say that x is an X.

which are windows. All the AUIOs are instances of the subclasses of class `VObject` and can be directly or indirectly used as building blocks of AOUIMS user-interfaces.

Class `VObject` provides the features needed to support views, and class `VCObject` additional features required to support control, such as the handling of mouse events. `Lines` and `Texts` are `VObjects`, and `Circles`, `Rectangles`, `Polygons`, and `Buttons` are `VCObjects`. A top-level user-defined class should be derived from `Toplevel`, and others from other AOUIMS classes.

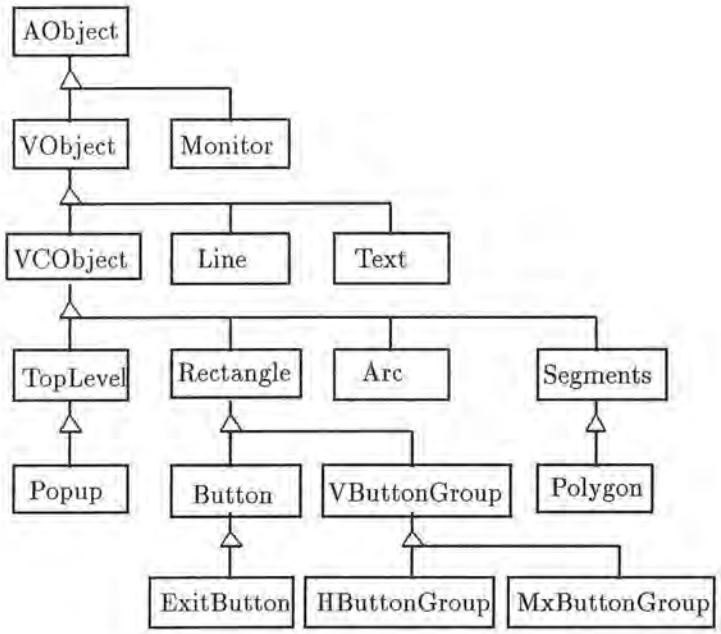

Figure 6: AOUIMS class hierarchy.

The hierarchy maintained on `VObjects` plays critical roles, as we explain later, in displaying or hiding groups of `VObjects` simultaneously and in processing mouse events. Any `VObject` can have children and become a *composite* AUIO. Any `VObject` other than a `TopLevel` can become a component of another `VObject`.

4.1 Control Variables and State Variables

We can activate some operations on `VObjects` by manipulating the values of their *control variables*, which are *condition variables* whose value changes can activate transition rules or `always` statements that perform the requested operations. The locations, sizes, and visibility of `VObjects`, for instance, are determined by control variables.

The location of the top-left corner of a `VObject` is defined by its attributes `xAbs` and `yAbs`, which are absolute coordinate values with respect to the top-level window. The location of a `VObject`, however, must be specified in terms of `xRel` and `yRel`, which are relative coordinates with respect to its parent `VObject`. When a user defines or changes the values of `xRel` and/or `yRel`, AOUIMS automatically computes its corresponding absolute coordinate values by using the function `alwaysCoord()`.

Control variables `myVisible`, `sysVisible`, `groupedVisible`, and `visible` are used to control the visibility of `VObject`. Variable `visible` ultimately determines the visibility of a `VObject`. If `visible` is `TRUE`, the `VObject` is displayed. Otherwise, it is not displayed. If a SAOS program wants to change the visibility of a `VObject`, it should manipulate its `myVisible`. Variable `sysVisible` is used by AOUIMS. The visibility of a `VObject` is further affected by the control variable `groupedVisible`.

In a graphical application, the *parent-child relationship* plays an important role, since many operations on a child object depend on the state of its parent object. For example, whether a child `VObject` should be displayed or not often depends on the visibility state of its parent `VObject`. AOUIMS provides control variables that determine whether the operations on each child object should depend on the state of its parent object.

4.2 Basic Classes

The most important basic AOUIMS classes relevant to the implementation of SAOSDB are `VObject` and `VCObject`. We now give brief descriptions of these classes.

4.2.1 Class VObject

Class `VObject` defines the view-related common behaviors of AUIOs. It is the superclass of such view-only classes as `Text` and `Line`. It is also the superclass of `VCObject`, which adds control to `VObject`. A `VObject` has the following interface variables.

Int xRel, yRel, width, height, xAbs, yAbs

These attributes determine the location and size of a `VObject`.

Int visible, myVisible, sysVisible, groupedVisible

These attributes control the visibility of a `VObject`.

Int inverse, myInverse, groupedInverse

These attributes dictate whether a `VObject` will be displayed in reverse video or not.

4.2.2 Class VCObject

Class `VCObject` defines the control aspects of AUIOs. It is the superclass of such classes as `Button`, `Rectangle`, and `Polygon`. It is also the superclass of `TopLevel` and `Popup`, which are the classes for windows. Since `VCObject` is a subclass of `VObject`, the capabilities provided for `VObjects` are applicable to `VCObjects`. For example, we can show or hide a `Popup` window simply by toggling its `myVisible` attribute.

Mouse events are passed along the component hierarchy of `VCObjects`. They are first passed from the top-most window (`TopLevel` or `Popup`) to lower-level `VCObjects`. A `VCObject` can *catch*, *pass*, or *discard* a mouse event. Furthermore, it can catch the event once passed if it is not caught by any lower-level `VCObjects`. This event pass mode is called `PassOrCatch`. We designed this event-passing mechanism by taking into consideration that a `VCObject` that must catch a mouse event often contains `VObjects` that may or may not catch mouse events.

When the user clicks a mouse button inside a `TopLevel` window, the AOUIMS receives a mouse event and sends it to the target `VCObject`. Although the default action for a mouse event is no action, it is possible to let the mouse event toggle the state of the control variable `selected` of the `VCObject`. This state change of `selected` causes function `_whenSelected()` to be activated. When `selected` becomes `TRUE`, `_whenSelected()` fills the area occupied by the `VCObject` and activates another function `myWhenSelected()`. Additional user-specified operations can be provided in this function, whose default operation is null.

5 Implementation

In this section, we describe the details of the implementation of SAOSDB. We first describe how the SaosDebugger graphical user interface is implemented. We then explain how the hierarchical structure of active objects are displayed and how the information in active objects is printed. Finally, the mechanism for tracing the executions of transition statements is explained.

5.1 Implementation of SAOSDB Graphical User Interface

Object saosDebugger, which is an instance of class SaosDebugger, is the top-level window for the debugger. Class SaosDebugger, shown in Fig. 7, contains viewModelButtonG, saosdbButtonG, and saosdbWindow as its member objects. Objects viewModelButtonG and saosdbButtonG are instances of class HButtonGroup (horizontal button group). Object saosdbWindow, which is an instance of class AORectangle, is used as the canvas where boxes representing active objects in the SAOS application program being debugged are displayed as a hierarchical tree.

Object saosdbWindow in SaosDebugger contains pointer rootObject that points to the AObjectView representing the root AObject in the SAOS application program. The root object is pointed to either by view or by model. The AObject pointed to by view is the TopLevel window of the application program.

Class definition of AObjectView is shown in Fig 8. An AObjectView is a box with additional lines inLine, outLine and vLine. The inLine is a line coming into the box from left. The outLine is a line coming out of the box from right. The vLine is a vertical line that connects the inLines of all the children AObjectViews to the outLine. The key component of an AObjectView is the pointer *subject* that points to the AObject in the SAOS application program. That AObject is associated with the AObjectView.

5.2 Displaying a Component Hierarchy

In this section, we describe how the active objects in the SAOS application program being debugged are displayed in a hierarchical tree structure.

Method whenButtonSelected() of class SaosDebugger is associated with saosdbButtonG so that when one of the buttons in saosdbButtonG is selected or deselected, whenButtonSelected()

```
class SaosDebugger : public TopLevel {
  public:
    Saosdebugger(char* title);
    ~SaosDebugger();
    virtual void initialize();
    void whenButtonSelected();          // invoked when one of the main
                                        // button group is selected
    void viewModelSelected();           // invoked when a view/model button
                                        // group is selected

    AORectangle   saosdbWindow;         // active objects display area
    HButtonGroup* saosdbButtonG;        // a matrix group button
    HButtonGroup* viewModelButtonG;     // point to a view/model
                                        // button group
    AObjectView*  rootObject;           // a view for a root active object
    ExitButton    exitButton;           // to exit program

  private:
    void setLabels();
};
```

Figure 7: Class `SaosDebugger`.

is activated. When the `Tree` button is selected, method `whenButtonSelected()` is activated, and SAOSDB enters the mode to display object structure. In this mode, if the box for an `AObjectView` is selected, function `onBoxSelected()` of class `AObjectView` is activated. After the descendants of the other siblings of the selected object are removed, the component hierarchy tree of the selected object is displayed. If the box for an `AObjectView` is deselected, the descendants of the selected object are removed.

Function `onBoxSelected()` will invoke only function `drawChildren()` if the selected object has no sub-tree and the sibling of the selected object has no sub-tree. Function `onBoxSelected()` will invoke both function `destroyChildren()` if the sibling of the selected object has a sub-tree and function `drawChildren()` if the selected object has no sub-tree. Function `onBoxSelected()` will invoke only `destroyChildren()` if the selected object already has its descendants displayed. Function `drawChildren()` which is shown in Fig. 9 is responsible for creating all the children views of the selected object. Function `destroyChildren()`, which is shown in Fig. 10, is responsible for removing all the subtrees of the selected object or all the subtrees of the sibling of the selected object from the canvas.

The two dimensional array of pointers `AObjectViewPtr[depth][position]` is used to keep

```
class AObjectView : public VCObject {
  public:
    AObjectView();
    ~AObjectView();
    virtual void initialize();
    virtual void onBoxSelected();      // which active object view is selected
    int          countChildren()       // number children used to calculate
                                       // the length of vertical line
    void destroyChildren(int);         // erase all children of this object

    AObject*     subject;              // its corresponce active object
    Line         inLine;               // line to left of AObject box
    Button       box;                  // a rectangle represent AObject view
    Line         outLine;              // line out of AObject box
    Line         vLine;                // vertical lines connect all children
    int          level;                // the level of this object
    int          number;               // the position of this object
                                       // in this level
    int          numberOfChildren();   // number of children are displayed

  private:
    void drawChildren(int, int, int, int, AObject*, int, int);
                                       // draw children of this object
};
```

Figure 8: Class `AObjectView`.

track of the AObjectView created on the canvas. Array element AObjectViewPtr[depth][position] points to the AObjectViews located at location (depth, position) in saosdbWindow. For example, the rootObject is pointed to by AObjectViewPtr[1][1]. If the rootObject has four children, each child is pointed to by AObjectViewPtr[2][i] where i = 1,..,4. The AObjectViewPtr[][] is used in drawChildren() and destroyChildren().

```
void AObjectView::drawChildren(int xRel, int yRel, int width,
      int height, AObject* subject,int level, int number) {
  if (subject) {
  AObjectView* saosdbObject = new AObjectView;  // create a view
  saosdbObject->subject = subject;
  saosdbObject->name = "saosdbObject";
  saosdbObject->xRel = xRel + 1;
  saosdbObject->yRel = yRel ;
  saosdbObject->width = width ;
  saosdbObject->height = height ;
  saosdbWindowPtr->insert(saosdbObject);
  saosdbObject->level = level;                    // object know its level
  saosdbObject->number = number;                  // and its number
  AObjectViewPtr[level][number] = saosdbObject; // keep track of a view
  };

  if (subject->sibling) {
    yRel = yRel + height * 5 / 4;
    subject = subject->sibling;                   // create its sibling views
    drawChildren(xRel,yRel, width, height, subject, level, number+1);
  };
};
```

Figure 9: Function drawChildren() defined for class AObjectView.

5.3 Printing Active Object Information

Every active object is an instance of class AObject or its derived class. Virtual function print() is declared in class AObject so that print() can be called for any instance of AObject and its derived classes. Since print() method for AOUIMS objects is supported by AOUIMS, the data of an AOUIMS object (application view) can be retrieved easily. However, in order for a programmer to have access to all the information in a user object in the application model, the print() method must be redefined for each user-defined class. When the Print button is selected, SAOSDB enters the print mode. In the print mode, if an AObjectView is selected,

```
void AObjectView::destroyChildren(int level) {
  int i;
  if ( level < MaxAObjectLevel - 1 ) {
  for(i=1; i < MaxAObjectChildren;i++)
    {
    if (AObjectViewPtr[level +1][i] != NULL)
      {
      delete AObjectViewPtr[level +1][i];  // destroy its view
      AObjectViewPtr[level +1][i] = NULL;
      };
    };
  outLine.visible = FALSE;
  vLine.visible = FALSE;
  destroyChildren(level+1);    // destroy all its children view
  }
  else
    return ;
};
```

Figure 10: Function `destroyChildren()` defined for class `AObjectView`.

function `onBoxSelected()` of class `AObjectView` is activated, and it prints all the information of the selected object on the standard output.

5.4 Program Tracing

SAOSDB supports two types of application program tracing. The tracing mode `Global Pause` allows the programmer to control the execution of every activated transition statement regardless to which object it is applied. The tracing mode `Object Pause` allows the programmer to control the execution of every activated transition statement applied to a particular object. We first describe the normal executions of activated transition statements and then explain the executions of transition statements when the programmer suspends the normal executions globally (`Global Pause`) or for a particular object (`Object Pause`).

The SAOS system maintains two main lists of activated transitions, one for activated trigger elements (`ATEs`) and the other for future events (`FEEs`). Activated transitions are either *system* activated transitions or *user* activated transitions. System activated transitions are mostly for graphical user interface manipulation. User activated transitions are those provided in the application programs. When a transition statement is activated, the activated transition is

enqueued either in an ATE or a FEE list. Activated transitions are executed on a first-come first-serve basis.

The main purpose of the debugger is to allow the programmer to control the executions of the user activated transitions. The system activated transitions are always required to be executed, but the executions of the user activated transitions must be able to be controlled by the programmer. Hence, they must be enqueued in the lists differently from those for the system activated transitions. We provide two lists for `ATEs`, one for the system activated transitions and the other for the user activated transitions. We do the same for `FEEs`.

5.4.1 Implementation of Global Pause and Global Next

`Global Pause` is a key feature provided for a programmer to suspend the execution of the user-activated transitions. In this section, we describe the mechanism for suspending the executions of activated transitions globally.

When the `GPause` button is selected, `whenButtonSelected()` of class SaosDebugger is activated. The user executions of activated transition statements in functions `executeATLs()` and `executeFELs()` are suspended immediately, and only system `ATEs` and `FEEs` activated transition statements are executed. The next user `ATE` activated transition statement is suspended from execution. The next user `FEE` activated transition statement is suspended as well. There is one important difference between executions of `ATEs` and `FEEs` activated transitions. All `ATEs` activated transition statements are executed without any condition, but `FEEs` activated transition statements are executed only when its `outTime` is greater than or equal to the `AOS time`. When the `GPause` button is selected, in order for `SAOSDB` to execute all system `FEE` activated transition statements, their `outTimes` are reducted by one if there are user `ATEs` activated transition statements in the list. Resuming of executions could be done when the programmer presses the `GNext` button. When `GNext` button is selected, function `executeATE()` is activated. The next activated transition statement in the list is the only activated transition statement to be executed. When the `Run` button is selected, the suspending of executions which is requested by the programmer is abandoned. All user `ATEs` activated transition statements and user `FEEs` activated transition statements are executed in `executeATLs()` and `executeFELs()` without suspending. The normal executions of the whole program are resumed at that point.

Code fragments for the executions of system and user `ATEs` and `FEEs` activated transitions are shown in Fig. 11, Fig. 13, and Fig. 12.

```
void executeATLs() {                          // execution of ATEs
    if((debugPause == 0) && (debugObjectPause == 0) &&
        aosATLs[teLevel].execute())           //execute all ATEs action
    else {
      if((debugPause == 1) || (debugObjectPause == 1)){
        aosATLs[AosSystem].execute();         // execute only system ATEs
};

      if((debugPause == 1) && (debugNext == 1)) {
        aosATLs[teLevel].executeATE();        // execute one ATE in ATL
      } // end if debugPause
      if((debugObjectPause == 1) && (debugObject != NULL )) {
        ATEPtr = (ATE*) aosATLs[teLevel].next1;  // a current ATE
        while ((ATEPtr->obj != NULL) && (ATEPtr->obj != debugObject))
                                              // execute the unselected objects
          aosATLs[teLevel].executeATE();

if(debugObjectNext == TRUE) {
          aosATLs[teLevel].executeATE();
  debugObjectNext = FALSE;
  teLevel++;            // to terminate this loop after execute one ATE
       }; // end if debugObjectNext
     }; // end if debugObjectPause
};
```

Figure 11: Function executeATLs() defined for the Global Pause command.

```
void ATL::executeATE() {      // execute one ATE in ATL
  ATEPtr = (ATE*) next1;      // a current ATE
  register ATE* ATEPtrTemp = ATEPtr;
  if (ATEPtrTemp && (ATEPtrTemp != (ATE*) this)) {
    ATEPtrTemp->unlink();
    ((ATEPtrTemp->obj)->*(ATEPtrTemp->pf))();
    if (debugLevel > 3) {
    cout << "my teLevel " << ATEPtr->teLevel << " my  ATE"
 << ATEPtr->seqNumber << " executed: "
 << ATEPtr->obj->pathName() << ":" << ATEPtr->text << endl;
    cout.flush();
    };
    delete ATEPtrTemp;
    ATEPtr = (ATE*) next1;    // will be used if debugObjectPause is on
    };
};
```

Figure 12: Function executeATE() defined for class ATL.

```
int executeFELs() {                    // execution of future events (FEEs)
  for (int level=0; level < NTELevels; level++) {
    temp = aosFELs[level].execute();        // execute all FEEs
};
```

Figure 13: Function `executeFELs()` definited for the `Global Pause` command.

5.4.2 Implementation of Object Pause and Object Next

The `Object Pause` option is used to control the executions of activated transitions of one particular object. The programmer can control the execution of a particular object after pressing the `ObjPause` button to enter the `object-pause` mode. In the `object-pause` mode, the programmer is responsible to select an object whose executions of user `ATEs` are suspended. In this section, we detail how suspending of executions of activated transition statements on a particular object is implemented.

When the `Object Pause` button is selected, `whenButtonSelected()` of class `SaosDebugger` is activated. If an `AObjectView` is selected, `onBoxSelected()` of class `AObjectView` is activated. All system `ATEs` and `FEEs` activated transition statements and user `ATEs` and `FEEs` activated transition statements of other objects are executed. The executions will be suspended eventually when the user activated transition statement for the selected object is encountered. Executions can be resumed when the programmer presses the `ObjNext` button. When the `ObjNext` button is selected, function `executeATLs()`, which is shown in Fig. 11, is activated which in turn activates function `executeATE()`, shown in Fig. 12. The activated transition statement of the selected object is suspended, but all other user activated transitions for the other objects will be executed until the next activated transition is for the selected object. When the `Run` button is pressed, the normal executions of the whole program is resumed.

6 Conclusion

SAOSDB is an object-based debugger for SAOS programs. It allows object-based, not statement-based, debugging for active objects. SAOSDB displays the overall structure of the application program as a hierarchical tree structure, and the user can use this structure to interact with active objects. One funtionality supported by SAOSDB is controlled executions of transition statements on active objects. The programmer can suspend and resume executions of transition statements. With this feature, the programmer can see the data of any active object as she single steps through the program. SAOSDB itself is constructed also as a SAOS on top of the SAOS kernel and AOUIMS.

7 Acknowledgements

I would like to extend my sincere gratitude to my major professor, Dr. Toshimi Minoura. Without his help and support, this work would not have been completed.

References

[BOOC91] Booch, G. *Object Oriented Software Design*, Benjamin/Cummings, 1991

[BUDD91] Budd, T. *An Introduction to Object-Oriented Programming*, Addison-Wesley, 1991

[CHOI91] Choi, S. and Minoura, T. *Active object system for discrete system simulation*, Proc. Western Simulation Multiconference, 1991, pp. 209-215.

[CHOI92] Choi, S. and Minoura, T. *User interface system based on active objects*, Proc. 2nd Symp. on Environments and Tools for Ada, Jan. 1992.

[GOLD80] Goldberg, A. and Robson, D. *Smalltalk-80: The language and its implementation*, Addison-Wesley, 1983

[KLEY88] Kleyn F. Minchael and Gingrich C. Paul. *GraphTrace - Understanding Object-Oriented Systems Using Concurrently Animated Views*, OOPSLA:191-205, 1988

[LIPP89] Lippman, S. B. *C++ Primer*, Addison-Wesley, 1989

[MINO93a] Minoura, T. and Choi, S. and Robinson, R. *Structural Active-Object Systems for Manufacturing Control*, Integrated Computer-Aided Engineering 1(2):121-136, 1993

[MINO93b] Minoura, T. and Pargaonkar, S. S. and Rehfuss, K. *Structural Active-Object Systems for Simulation*, OOPSLA:338-345, 1993

[MINO93c] Minoura, T. and Choi, Sungoon. *Active-Object User Interface Management System*, Proc. Tools USA 93, Prentice Hall:303-317, 1993

[MINO93d] Minoura, T. and Choi, Sungoon. *Structural Active-Object Systems Fundamentals*, Dept. of CS, Oregon State University, 93-40-04, 1993

[MINO93e] Minoura, T. and Choi, Sungoon and Pareddy, Raghava *Environments for Active-Object Systems*, Dept. of CS, Oregon State University, 93-40-04, 1993

[MEYE88] Meyer, B. *Object-Oriented Software Construction*, Prentice-Hall, 1988

[PURS91] Purchase A. Jan and Winder L. Russel *Debugging tools for object-oriented programming*, Journal of object-oriented programming:10-27, June 1991

[RUMB91] Rumbaugh, J., et al. *Object-oriented modeling and design*, Prentice-Hall:10-27, 1991

[STRO86] Stroustrup, B. *The C++ Programming Language*, Addison-Wesley, 1986.