Structural Active Object Systems for Mixed-Mode Simulation

Shirish S. Pargaonkar
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-4602
shirish@mist.cs.orst.edu

Abstract

A *structural active-object system* (SAOS) is a *transition-based object-oriented* system suitable for the design of various concurrent systems. A SAOS consists of a collection of interacting *structural active objects* (SAOs) whose behaviors are determined by the *transition statements* provided in their class definitions. Furthermore, SAOs can be *structurally* and *hierarchically* composed from their component SAOs like hardware components. These features allow SAOs to model components for circuit simulation more naturally than *passive objects* used in ordinary object-oriented programming, including digital, analog, and mixed-mode simulation. Each hardware object such as an AND gate or an integrator can be represented as a SAO. In addition, *structural* and *hierarchical* composition allows us to build complex components from elementary components. Prototype simulation programs with graphical user interfaces have been developed as SAOS programs for digital, analog, and mixed-mode circuit simulation.

Key Words and Phrases: digital simulation, analog simulation, mixed-mode simulation, active-object system, concurrent object-oriented programming, structural composition, hierarchical composition, software IC, graphical user interface

1 Introduction

Object-oriented programming (OOP) [GOLD80, MEYE88, STRO86] is making fundamental changes in software development. Such features as *encapsulated classes*, *inheritance*, and *polymorphism* provided by OOP allow us to implement highly modular reusable software components. Furthermore, since objects which embody state and behavior resemble to real-world objects better than traditional software modules, object-orientation provides a suitable framework for software development [BOOC91, RUMB91].

A *structural active-object system* (SAOS) is an *object-oriented concurrent* system using *transition (production) rules*, *equational assignment statements*, and *event routines* for its behavior description. Production systems have been known to be suitable for various concurrent systems that require flexible synchronization [ZISM78]. The SAOS approach integrates object-orientation and production rules. The key mechanism used by SAOSs is *structural* and *hierarchical* composition of *structural active objects* (SAOs). Structural/hierarchical composition allows SAOs to be constructed from their component SAOs like hardware objects. Note that hardware objects are active autonomous objects. Structural and hierarchical composition is universally used in the design and implementation of such complex electronic and mechanical devices as VLSI chips and automobiles.

The behavior of each SAO is determined in the *transition statements* provided in the *class* definition of that SAO. Each transition statement is a *transition rule*, which is a *condition-action* pair, an *equational assignment statement*, or an *event routine*. Equational assignment statements maintain simple invariant relationships among SAO states. Event routines are activated by messages. Since the behaviors of SAOs are determined by user-programmed transition statements, classes for new types of components can be easily implemented. Even such devices as recorders can be constructed as SAOs.

One key feature of SAOs is that the transition statements provided for each SAO can access, besides the state of that SAO, the states of the other SAOs known to the SAO through its *interface variables*, thus realizing inter-object communication. We can establish desired connections among SAOs by binding proper SAOs to interface variables. Interface variables

are like *terminals* of hardware components, and they are crucial for structural composition. The SAOS approach primarily uses structural composition of SAOs whereas conventional OOP uses procedural interfaces provided for passive objects. SAOs can be structurally and hierarchically composed through interface variables since each SAO encapsulates its portion of control.

The idea of *active* objects originated with the first object-oriented language SIMULA [BIRT73], where active objects were simulated by coroutines. Several object-oriented concurrent systems have been designed since then. *Actors* introduced active computational agents that carry out their actions in response to incoming messages [AGHA86]. *ABCL* objects [YONE87] and *Emerald* objects [BLAC86] may be sequential processes that exchange messages among them.

The major goal of the SAOS approach is to provide a single framework that can be used throughout a software lifecycle. For this purpose, SAOSs are graphically represented by *SAOS diagrams*. SAOS diagrams can be used as design documents from which executable code can be generated and as user interfaces.

The SAOSs are written in a description language called Structural Active Object System Descriptive Language (SAOSDL). This language can be compared with VHDL which is an existing description language for digital circuits.

In VHDL, each operation of a discrete system is referred as a process. In SAOSDL, an operation is represented by a transition statement. In VHDL, a signal is used to handle communication between processes. Signals define data pathways between processes on which values are passed. VHDL provides means for a process to express sensitivity to the value of a data pathway. These data pathways are called sensitivity channels. There is a type associated with every signal. Thus processes with different signals or data pathways can not communicate. In SAOSDL, communication takes place between objects. Communication between two objects is achieved when an object shows an interest in a state variable of another object by adding an trigger element to the trigger list of that state variable. This is similar to the VHDL which provides means to a process to express sensitivity to the value of a data pathway.

VHDL is not an object-oriented language. Although a hardware component is represented

as an entity in VHDL, we can not derive a subclass from an entity to define a new entity. SAOSDL is an objet-oriented language. Thus it allows users to define a new component by subclassing an existing component. A common feature between VHDL and SAOSDL is that they both support structural composition which enables user to create new components from existing subcomponents.

In VHDL a process is activated only when the value on the sensitivity channel changes. Similarly in SAOSDL, a transition statement within an object is executed when the value of the state variable to which an object has added a trigger element changes.

An object in SAOSDL is referred to as an entity in VHDL. An entity consists of an entity declaration and architecture body. Entity declaration provides the external view of the component. It describes what can be seen from the outside, including the component's ports. Architecture body provides the internal view. It describes the behavior of the structure of the component. Similarly in SAOSDL, the class definition of an object consists of interface part and body part. Interface part consists of member variables in the class definition of a component and body part consists of the functions provided to define behavior of the component.

Another significant difference between the two is that VHDL is a purely textual language whereas SAOSDL comes with graphical representations of components[LIPS90].

Design automation tools have become essential to current engineering activities. In the area of digital systems design, a host of tools have been developed for schematic capture, layout, design-rule checking, and simulation[BLOO87, DAVE86]. The SAOS approach can integrate all of these activities. A SAOS graphical editor can be used to create SAOS diagrams at the schematic capture stage. Design-rules can be enforced by transition statements. Furthermore, SAOS diagrams can be used as user-interfaces during simulation. In fact, our prototype SAOS graphical editor allows the user to activate a system being layed out or being modified before the design is complete.

Blending analog and digital simulation is generally believed to be difficult[GOER88], and the analog and digital portions of a system are often designed separately. Since both analog and digital components such as logic gates, flip-flops, and integrators can be represented as SAOs,

mixed-mode simulators can be easily implemented as SAOSs. Besides modeling functionalities of components, SAOSs provide dynamic (animated) graphical user-interfaces. SAOS user-interfaces are supported by the Active Object User-Interface Management System (AOUIMS) [CHOI92b]. In fact, a SAOS program can be constructed by *pick-and-place* operations with a graphical editor.

Section 2 introduces the SAOS approach for digital-circuit simulation by using simple D-latch and D-flipflop circuits as examples. An analog circuit simulator as a SAOS is described in Section 3. Section 4 discusses the details of mixed-mode simulation. Finally, Section 5 concludes the report.

2 Digital Circuit Simulation

In this section, we first discuss how a DLatch can be implemented as a SAOS program and explain its translated C++ code. We then show an edge-triggered D flip-flop implemented as a SAOS program.

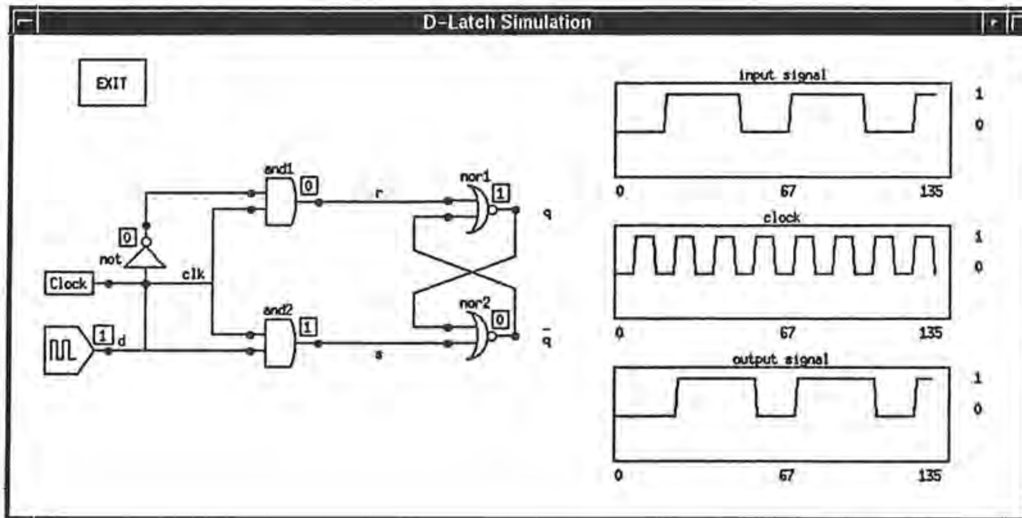

Figure 1: DLatch.

Fig. 1 shows a DLatch. The circuit consists of an SR latch composed of two `Nor` gates `nor1` and `nor2`, and a clock gating circuit composed of two `And` gates `and1` and `and2`. The clock signal *clk* and the data signal *d* are input signals, and the complementary signals q and $\bar{q}$ are output signals. If, at some time, $d = 0$ and *clk* is high, we have $r = 1$, $s = 0$, $q = 0$, and $\bar{q} = 1$. Similarly, if $d = 1$ and *clk* is high, we have $r = 0, s = 1, q = 1$, and $\bar{q} = 0$. If *clk* is low, the SR latch is disconnected from *d*.

Fig. 2 gives the SAOS class definition for a DLatch. In general, a SAOS class definition consists of the following three parts:

1. An *interface* part preceded by the keyword `public:` as in C++ specifies the *input ports* and the *output ports*. An input port can be connected to an output port of another component as an input pin of a hardware component can be connected to an output pin of another hardware component.

```
class DLatch {                    // class definition for a D-latch
  public:
    Int*  clk;                    // input port for clock
    Int*  d;                      // input port for D-latch input
    alias q    = nor1.output;     // output port for D-latch output
    alias qNot = nor2.output;     // output port for negated D-latch output
  private:
    Not not   with {input = d};
    And and1  with {input1 = &not.output, input2 = clk};
        and2  with {input1 = clk, input2 = d};
    Nor nor1  with {input1 = &and1.output, input2 = &nor2.output};
        nor2  with {input1 = &nor1.output, input2 = &and2.output};
}
```

Figure 2: SAOS class DLatch.

2. A *class body* contains a set of *instance variables.* These instance variables represent the subcomponents of an instance of the class.

3. A *behavior description* specifies the functionality added at this class level.

The `interface` part of class `DLatch` specifies that a `DLatch` has *input ports* `cl` and `d`, and that instance variables `output` of gate `nor1` and `nor2` become the output ports `q` and `qNot`, respectively.

The body of class `DLatch` consists of one instance `not1` of class `Not`, two instances `and1` and `and2` of class `And`, and two instance `nor1` and `nor2` of class `Nor`. These components are statically interconnected by `with` clauses. For example, input port `input` of gate `not` is connected to an input port `d`. Input ports `input1` and `input2` of gate `and1` are bound to `output` of gate `not` and to the `clk`, respectively, and so on.

In the case of class `DLatch`, the behavior description is empty. Its behavior is determined completely by its interconnected subcomponents. In general, a SAOS class can include a behavior description as we see in later examples.

We now show the definitions of the classes used by class `DLatch`. The class `Gate` shown in Fig. 3 is the base class for such gate classes as `And`, `Nor`, and `Not`. It defines the output value of a gate as the output port `output`.

```
class Gate {              // base class for classes And, Nor, Not, etc.
  public:
    Int output;           // gate output value, output port
}
```

Figure 3: SAOS base class Gate.

```
class And : public Gate {  // class for AND gates
public:
    Int *input1, *input2;  // input ports for input values
  private:
    always output = (input1->output && input2->output);
}
```

Figure 4: SAOS class And.

The class And in Fig. 4 is derived from class Gate. It inherits output port output from class Gate. In addition, it has input ports input1 and input2 as interface variables, which can be bound to the output ports ouptut of other gates. This SAOS class does not use any subcomponents. The always statement defines the behavior (functionality) of an And gate. Whenever any one of the input values changes, the always statement updates the value of output to the outcome of logical AND operation on the two input values.

```
class Nor : public Gate {  // class for NOR gates
  public:
    Int *input1, *input2;  // input ports for input values
  private:
    always output = not (input1->output || input2->output)
}
```

Figure 5: SAOS class Nor.

The class Nor shown in the Fig. 5 is defined in a similar way. Class Nor differs from class And only in its behavior specification defined by its always statement. The definition of the class Not is shown in Fig. 6.

A SAOS program discussed above is translated into a C++ program. Our prototype SAOS translator can perform this translation. However the C++ code we show here was hand-coded.

```
class Not : public Gate {  // class for NOT gates
  public:
    Int *input;            // input port for input value
  private:
    always output = not (input->output);
}
```

Figure 6: SAOS class Not.

In order to illustrate the operation of a structural active-object (SAO), we explain the details of the class And translated into C++.

```
class Gate : public Segments {  // base class for classes And, Nor, Not, etc.
 public:
   Int  output;                 // output value, output port
   Gate(int n) : Segments(n) {}; // constructor
};
```

Figure 7: C++ class definition of Gate.

Fig. 7 shows the C++ definition of class `Gate`. Its parent class `Segments` provides graphical representation for gates. The shape of each gate type is formed by a set of line segments.

```
class And : public Gate {            // class for AND gates
 public:
   Int* input1;                      // input port 1
   Int* input2;                      // input port 2
   And() : Gate(4) {};               // constructor
   virtual void initialize();        // initialization routine
           void whenInputChanged(); // functionality
}
```

Figure 8: C++ class definition of And.

Class `And` shown in Fig. 8 is derived from class `Gate`. The type `Int` designates a *condition variable* for an integer. A condition variable maintains a list of pointers to functions, called a *trigger list*. Whenever the value of a condition variable is updated, the functions pointed to by the elements of the trigger list are executed. In the case of class `And`, the function `whenInputChanged()` shown in Fig. 9 is activated whenever any of the input values changes.

```
void And::whenInputChanged() {
  output = (int) *input1 && (int) *input2;  // compute output
}
```

Figure 9: Functionality definition of an And gate.

In this way, the behavior specified by the always statement is implemented.

```
void And::initialize() {
  Gate::initialize();  // base class initialization
  // trigger setups
  PROC pf = PROC (&And::whenInputChanged));
  input1->tl.addTE(this, pf, "whenInputChanged()");
  input2->tl.addTE(this, pf, "whenInputChanged()");
}
```

Figure 10: Initialization function of an And gate.

The major task of function initialize() shown in Fig. 10 is to set up triggers for function activations. Since function whenInputChanged() must be activated whenever any of the input values changes, a trigger is added to each of the inputs by an addTE() function.

```
class DLatch : public Segments {
 public:
   Int* clk;                    // input port for the clock
   Int* d;                      // input port for the signal
   Int  q;                      // output value at Q
   Int  qNot;                   // output value at Q
   DLatch() : Segments(0) {};   // constructor
   void initialize();           // initialization
 private:
   VNot not1;                   // Not gate with vertical orientation
   And  and1, and2;             // And gates used for gating circuit
   Nor  nor1, nor2;             // Nor gates used for SR flip-flop
}
```

Figure 11: C++ class definition of a DLatch.

We now show the C++ definition of class DLatch in Fig. 11. First, interface variables such as clk, d, q and qNot are provided. Thus, an external clock can be connected to clk, an input external signal can be connected to d, and output q of DLatch can be connected as an input of

another external component. Then the subcomponents that comprise a DLatch are declared. The class definition also includes the declaration of function initialize(). Its major function is to interconnect subcomponents as shown in Fig. 12.

```
void DLatch::initialize() {
 Segments::initialize();          // parent class initialization

 not1.input  = d;                 // provide connections
 and1.input1 = &not1.output;
 and1.input2 = clk;
 and2.input1 = d;
 and2.input2 = clk;
 nor1.input1 = &and1.output;
 nor1.input2 = &nor2.output;
 nor2.input1 = &and2.output;
 nor2.input2 = &nor1.output;
}
```

Figure 12: Initialization of a DLatch.

```
class DLatchSys : public TopLevel {
 public:
   DigiClock  clk1;                       // a clock
   SigGen     sgn1;                       // a signal generator
   DLatch     dlch1;                      // a DLatch
   DigiRecorder recrd1, recrd2, recrd3; // various recorders

   DLatchSys(char* n) : TopLevel(n) {};
   void initialize();
}
```

Figure 13: C++ class definition of circuit containing a DLatch.

Fig. 13 shows the definition of a circuit that was used to test a DLatch. In order to construct this SAOS program, we have defined three additional classes DigiClock, SigGen, and Recorder. A signal generator that generates square wave with the user-specified duty cycle is defined by C++ class SigGen. Class DigiClock is identical to class SigGen. To plot a signal of type Int, a DigiRecorder can be used. If an analog signal, which is of type Float, is to be plotted, a Recorder must be used. Fig. 14 specifies the interconnections among these components as shown in Fig. 1.

```
// external clock output is connected clock input port
dlch1.clk    = &clk1.output;
// signal generator output is connected to signal input port
dlch1.d      = &sgn1.output;
// external clock output is connected to the clock input port
recrd1.input = &clk1.output;
// signal generator output is connected to recorder input
recrd2.input = &sgn1.output;
// output of the D latch connected to recorder input
recrd3.input = &dlch1.q;
```

Figure 14: Interconnections among the components in DLatchSys.

It is important to emphasize that we have constructed a recorder also as a SAO. To connect a component to a recorder, its output is equated to the input of a recorder. Its behavior is implemented by a function `aosTimeChanged()` which is executed for every SAOS time change. After a wave form reaches the end of a recorder time span, the time span is shifted halfway and plotting starts from the middle of the recorder.

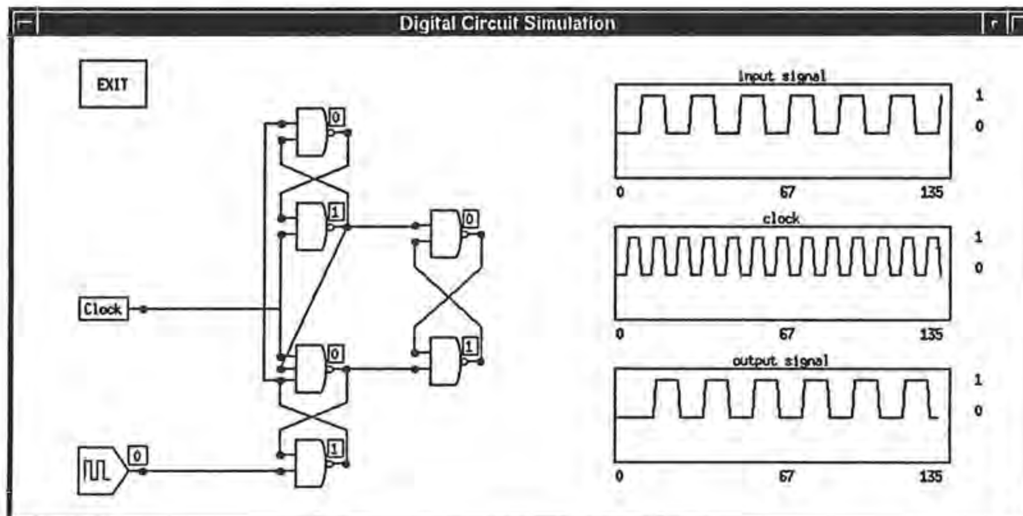

Figure 15: D Flip-Flop.

We have also implemented a negative-edge-triggered D flip-flop as shown in Fig. 15.

The C++ class definition of `DFlipFlop` is shown in Fig. 16. It consists of six NAND gates and ports `clk` for external clock input, `d` for external signal input, and `output` for the output. A NAND gate is defined by the class `Nand`. An user can define a NAND gate with any number

```
class DFlipFlop : public Segments {
  public:
    Int*  clk;                 // input port for clock
    Int*  d;                   // input port for signal
    Int   output;              // D flip-flop output
                               // constructor
    DFlipFlop() : Segments(0), n1a(2), n1b(2), n2a(3), n2b(2),
                               n3a(2), n3b(2) {};
    void initialize();         // initialization routine
    void whenInputChanged();   // functionality
  private:
    // various NAND gates used to build a D flip-flop
    Nand  n1a, n1b, n2a, n2b, n3a, n3b;
}
```

Figure 16: C++ class definition of a D flip-flop.

of input ports by specifying it in its constructor. For example, as shown in Fig. 16, we have defined a NAND gates n1a with two input ports and n2a with three input ports. Although, we have defined a C++ class And shown in Fig. 8 with two input ports, it can be redefined or subclassed to accept variable number of inputs similar to the class Nand. The D flip-flop works as follows: The data present when clock is high is transferred to the output of the flip-flop when clock becomes low. When clock makes a transition from 1 to 0, the output becomes 0 if input signal is 0, and output becomes 1 if input signal is 1.

```
class DFFCircuit : public TopLevel {
  public:
    DigiClock clock;       // clock
    SigGen    siggen;      // signal generator
    DFlipFlop dff;         // D flip-flop
                           // recorders to plot timing diagrams
    Recorder  recorder1, recorder2, recorder3;

    void setWindowSize();
    DFFCircuit(char *n) : (n);
    void initialize();
}
```

Figure 17: C++ class definition for a circuit containing D flip-flop.

Fig. 17 shows the definition of a circuit which has a D flip-flop.

```
// external clock output is connected clock input port
dff.clk    = &clock.output;
// signal generator output is connected to signal input port
dff.d      = &siggen.output;
// external clock output is connected to the clock input port
recorder1.input = &clock.output;
// signal generator output is connected to recorder input
recorder2.input = &siggen.output;
// output of the D flip-flop connected to recorder input
recorder3.input = &dff.output;
```

Figure 18: Interconnections among the components of a circuit containing a D flip-flop.

Fig. 18 shows how these components are connected to construct the circuit. Similar to the `DLatch`, the clock input of the D flip-flop is connected to the digital clock and signal input is connected to the signal generator. Also, the three recorders are connected to the clock, signal generator, and output of the D flip-flop.

Structural composition, a key feature of SAOS, proves very useful in building complex components and circuits. This was demonstrated by building components such as D latch and D flip-flop from subcomponents and using them to construct circuits.

3 Analog Circuit Simulation

We now discuss how an analog simulator can be implemented as a SAOS. The major difference between an analog simulator and a digital simulator is that integrators used by an analog simulator must recompute their output values when system time changes as well as their input values change. The circuit shown in Fig. 19 simulates a system represented by the differential equation: $\ddot{y} = 2 * (5 - y - 0.5\dot{y})$.

Figure 19: Analog simulator.

```
class AnalogSim : public TopLevel { // analog simulator class
 public:
   FloatSource  src;              // source providing reference value 5.0
   Adder        adr;              // an adder to add three input values
   RScaler      scl1, scl2;       // reverse coefficient scalers
   Scaler       scl3;             // coefficient scaler
   Integrator   integ1, integ2;   // integrators
   Recorder     recrdr;           // analog signal recorder

   AnalogSim(char* n) : TopLevel(n), adr(3) {};    // constructor
   void initialize();
}
```

Figure 20: Analog simulator.

Fig. 20 gives the SAOS class description of the analog simulator. It uses a reference-value

source `src`, an analog signal adder `adr`, three scalers `scl1`, `scl2`, and `scl3`, two ingegrators `integ1` and `integ2`, and an analog signal recorder `recrdr`. These components are connected as specified in Fig. 21. The output of a component is connected to the input of another component by assigning the address of the output to the pointer variable pointing to the input. The initial values of the `Integrators` are assigned as `integ1.initVal = 0` and `integ2.initVal = 0`.

```
adr.input[0]  = &src.output;    // source output connected to input0 of adder
adr.input[1]  = &scl2.output;   // scaler2 output connected to input1 of adder
adr.input[2]  = &scl3.output;   // scaler3 output connected to input2 of adder
scl1.input    = &adr.output;    // adder output connected to scaler 1
integ1.input  = &scl1.output;   // scaler1 output connected to integrator input
integ2.input  = &integ1.output; // integ1 output connected to integ2 input
scl2.input    = &integ1.output; // integrator1 output connected to scaler2 input
scl3.input    = &integ2.output; // integrator2 output connected to scaler3 input
recrdr.input  = &integ2.output; // integrator2 output connected to recorder
```

Figure 21: Interconnections among the components.

When the output value v of a component changes, all the components to which it is connected receive triggers to recompute their output values immediately by using the new value v. These components may further send triggers to other components to recompute their output values, propagating the changes. This is accomplished by making the output of each component a variable of type `Float`. A `Float` variable maintains a `float` value and a list of the pointers to the functions to be executed when that value changes. In the case of an `Integrator`, its output is recomputed whenever system time `aosTime` changes as well as when its input value changes.

We now describe how various components used by analog simulators are implemented. Fig. 22 gives the definition of C++ class `Integrator`. Variable `input` is a pointer to `output` of another analog component, and variable `output` is the output of an `Integrator`. An `Integrator` maintains two successive input values to be used by the *trapezoidal integration method.* Variable `newInVal` maintains the input value at time `inputTime`, and variable `oldInVal` maintains the input value at time `inputTime` -1. Similarly, variable `sum` maintains the integrated value at time `inputTime`, and variable `oldSum` maintains the integrated value at time `inputTime` -1.

```
class Integrator : public Segments {
 public:
   Float* input;           // input port pointer
   int    inputTime;       // sampling time of newInVal
   float  newInVal;        // input value at inputTime
   float  oldInVal;        // input value at inputTime - 1
   float  sum;             // integrated value
   float  oldSum;          // integrated value at inputTime - 1
   Float  output;          // output port
   Float  initVal;         // initial value
   Integrator();           // constructor
   void initialize();      // initialization routine
   void integrate();       // performs integrations
}
```

Figure 22: C++ class Integrator.

```
void Integrator::integrate() { // integrator functionality definition
  if (aosTime && aosTime == inputTime + 1) {   // new system time
    oldSum = sum;                // save integrated value at aosTime - 1
    oldInVal = newInVal;         // move old new input value to old input value
    inputTime = aosTime;         // update input time
  }
  newInVal = *input;             // read current input value
  if (aosTime) {                 // not initiation time
    float deltaOut  = (oldInVal + newInVal) / 2.0 * DELTA; // compute delta
    sum = oldSum + deltaOut;   // perform integration
    if (absf(sum - output) > EPSILON)  // propagate output value
      output = sum;                    // if change is significant
  }
};
```

Figure 23: Functionality definition of an Integrator.

Fig. 23 shows the implementation of the integration algorithm based on the trapezoidal method. The function `integrate()` is activated either when `aosTime` is incremented or when the input value of the `Integrator` changes. The amount of change to the integrated value is computed by the line:

```
float deltaOut  = (oldInVal + newInVal) / 2.0 * DELTA;
```

At any given `aosTime`, function `integrate()` may be activated more than once. In each successive activation, presumably, a more accurate integrated value `sum` will be recomputed. Note that the incremental integrated value `deltaOut` is added to `oldSum`, which is the integrated value at `aosTime` -1, and not to `sum`.

If the computed integrated value `sum` is always set to `output`, `integrate()` may be activated too many times at each `aosTime` if a simulated circuit has a feedback loop. In order to prevent this problem, if the difference between `sum` and `output` is within the specified limit `EPSILON`, `output` of `Integrator`, is not set to `sum`. Thus the value of `EPSILON`, set by the user, controls the time for convergence and the accuracy of the integrated value of an `Integrator`.

```
class Adder : public VCObject {
 public:
   Float** input;                 // array of input ports
   Float   output;                // output value
   int     nInputs;               // number of inputs, specified by user

   Adder(int);                    // constructor
   void initialize();             // initialization routine
   void whenInputChanged();       // perform addition
}
```

Figure 24: C++ class definition Adder.

The class `Adder` shown in Fig. 24 is used to create `Adders`, each of which computes the sum of its multiple input values. The number of the inputs must be specified by the user as the parameter of the constructor. The function `whenInputChanged()` shown in Fig. 25 performs the addition of the `input` values. Therefore, it must be added to the *trigger list* of the `Float` variable pointed to by each `input` and activated whenever the value of that variable changes.

```
void Adder::whenInputChanged() { // functionality of an Adder
  float sum = 0;
  for (int i = 0; i < nInputs; ++i)
     sum += ((float) *input[i]);
  output = sum;
};
```

Figure 25: Functionality definition of an Adder.

A `Scaler` provides a multiplication factor `multFactor` whose value can be set by the user. The value can be positive or negative. The value of `output` is a product of `mulFactor` and the value of the `Float` variable pointed to by `input`. As in the class `Adder`, the value of `output`, is recomputed whenever the input value changes. Class `RScaler` is a subclass of the `Scaler`. The only difference between these two classes is their graphical representations. In the example shown in Fig. 19, the multiplication factors of the scalers are set as `scl1.multFactor = - 1.0`, `scl2.multFactor = - 0.5`, and `scl3.multFactor = 2.0`.

The class `FloatSource` supplies a constant `float` value. It is derived from the parent class `Source`. A `FloatSource` does not have an input port. The user can specify the value supplied by the `FloatSource` as `src.output = 5.0`.

A `Recorder` is used to plot the output value generated by `Integrator integ2`. It is also implemented as a SAO in the same way as a `DigiRecorder`.

4 Mixed-Mode Circuit Simulation

In the preceding sections, we discussed how digital and analog simulators can be implemented as SAOS programs. As we explained in the implementation of analog components, transition statements defining the behaviors of SAOs can be activated both by events and system time changes. Therefore, it is straightforward to implement mixed-mode simulators as SAOSs, where digital components are triggered by events, and analog components primarily by changes of the system time.

In this section we show an example of a mixed-mode simulator. The circuit shown in Fig. 26 consists of an analog part, a digital part, and an interface between these two parts. The analog part consists of a sine-wave generator, and the digital part consists of a modulo-16 ripple counter. A comparator is used as the interface between the analog and digital parts.

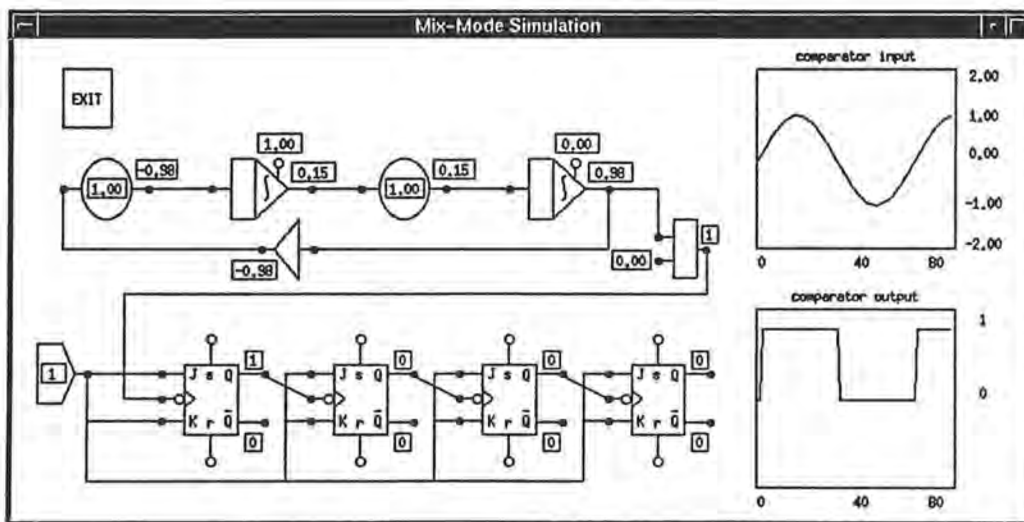

Figure 26: Mixed-mode simulation circuit.

Fig. 27 gives the description of the circuit. The analog part uses instances of the classes `Integrator`, `Scalar`, and `Inverter`. The digital part uses instances of the classes `JKFlipFlop` and `IntSource`, and the interface part uses instance of the class `Comparator`.

The comparator used as the interface between the analog and digital parts functions as follows. A comparator provides its output value at two fixed levels, logical 0 and 1, depending on whether its input value v_i is larger or smaller than the reference value v_r. In our example,

```
class MixSimSys : public TopLevel {
  public:
    // analog part
    Integrator   integ1, integ2;
    Scaler       scl1, scl2;
    Inverter     inv;
    // interface between analog and digital part
    Comparator   comp;
    // digital part
    IntSource    src;
    JKFlipFlop   jkf1, jkf2, jkf3, jkf4;
    // recorders
    Recorder     recrd1;
    DigiRecorder recrd2;
    MixSimSys(char *n) : TopLevel(n) {};
    void initialize();
}
```

Figure 27: Class definition for a mixed-mode simulation circuit.

when the output value of the sine wave generator crosses 0 and becomes positive, the output of the comparator changes from 0 to 1. When the transition occurs in the opposite direction, the output value changes from 1 to 0. Since we assume that the `Comparator` is ideal, the range of uncertainty is null.

```
class Comparator : public Segments {
 public:
   Float*       input;                     // input port
   Int          output;                    // output value
   Float        refVal;                    // reference value for comparison
   Comparator(): Segments(0) {};           // constructor
   void initialize();                      // initialization
   void whenInputChanged();                // functionality
};
```

Figure 28: C++ class Comparator.

As you can see in the definition of C++ class `Comparator`, its input port is a pointer to a variable of type `Float`, and its output port is a variable of type `Int`. Therefore, the output of an analog component can be connected to the input port, and the output port to an input port of a digital component. The variable `refVal` stores the reference value v_r of a `Comparator`,

whose behavior is shown in Fig. 29. The reference value of a `Comparator` can be set by the user as `comp.refVal = 0`

```
void Comparator::whenSignalChanged() {
  if (*input > refVal)
     output = 1;       // output is high when Vi is above Vr
  else
     output = 0;       // output is low when Vi is below Vr
}
```

Figure 29: Functionality of a comparator.

Before we discuss how the digital part of this circuit works, let us look at the JK flip-flop and how its behavior is implemented. Class `JKFlipFlop` defines a JK flip-flop. Fig. 30 shows the implementation of the behavior of the JK flip-flop in the function `whenClockChanged()`.

```
void JKFlipFlop::whenClockChanged() {
  if (*clk) {                    // outputs change only when clock is low
   switch (*inputJ) {
    case 0:
     switch (*inputK) {          //  J = 0, K = 0
      case 0:                    //  so Q and QNot do not change
       break;
      case 1:                    // J = 0, K = 1
       outputQ = 0;              // so Q = 0 and QNot = 1
       outputQNot = !(outputQ);
     }
    case 1:
     switch (*inputK) {
      case 0:                    // J = 1, K = 0
       outputQ = 1;              // so Q = 1 and QNot = 0
       outputQNot = !(outputQ);
       break;
      case 1:                    // J = 1, K = 1
       outputQ = !(outputQ);     // so Q and QNot inverse
       outputQNot = !(outputQ);
     }
   }
  }
};
```

Figure 30: Functionality definition of a JK flip-flop.

The modulo-16 ripple counter forms the digital part of the circuit. Four JK flip-flops are connected in the toggling mode with $J = K = 1$. The input signal whose cycles are to be counted is applied to the clock input `clk` of the first flip-flop. The output $\bar{q}$ of the first flip-flop is connected to the clock input `clk` of the second flip-flop, the output $\bar{q}$ of the second flip-flop is connected to the clock input `clk` of the third flip-flop and so on. The JK flip-flop changes its state when and only when the its clock input value changes from 1 to 0. During the initial state of the counter, output values q of all JK flip-flops are 0. The sequence of input pulses takes the counter through all possible $2^4 = 16$ states so that after the sixteenth clock pulse, counter returns to its initial state.

The class `IntSource` supplies a constant value of 0 or 1 specified by the user. It is derived from the parent class `Source`. A `IntSource` does not have an input port.

The analog part consisting of `Integrators` `integ1` and `integ2` generates a sine wave of amplitude 1. It is possible to change the amplitude of the wave form by changing the initial conditions of the `Integrators` and the frequency of the wave form by changing the `multFactors` of the `Scalers`.

The class `Inverter` inverts the polarity of analog signal. This is implemented by multiplying its input value by -1 to generate an inverted output value.

Fig. 26 shows two recorders. The top recorder plots the sine wave and the bottom recorder plots the output of a comparator. It is possible to observe and compare the outputs of each JK flip-flop by connecting them to respective recorders.

5 Conclusions

The *structural active-object system* (SAOS) approach provides a new framework for developing digital, analog, and mixed-mode simulation programs. A SAOS can be constructed from its component *structural active objects* (SAOs) by *structural* and *hierarchical* composition. SAOs are self-contained and active, and their behaviors are defined by the *transition rules*, **always** *statements*, and *event routines* provided in their class definitions. They interact with other SAOs connected through their *interface variables*, which correspond to terminals of hardware components.

We have successfully implemented simulators for digital, analog, and mixed-mode circuits as SAOS programs, building such basic digital circuit components as AND, OR, NOT, NOR, EXOR, NAND gates, D flip-flops and JK flip-flops and such analog components as integrators and comparators. Although the components we have designed are ideal, future versions can be made more realistic by including such features as gate delays.

We have developed various prototype SAOS programs and realized the following benefits of the SAOS approach.

1. Structural and hierarchical composition allows us to construct complex software components from their basic building blocks as though they were indeed hardware components. The resultant system descriptions, especially in *SAOS description language* (SAOSDL), closely reflect the circuits simulated and are very concise.

2. SAOS programs can be written either in SAOSDL or in C++. In defining classes for new SAOs, these languages allow us to use such features of object-oriented languages as inheritance and virtual functions. These features are not available in other hardware description languages such as VHDL.

3. It is easy to provide animated graphical representations of the systems simulated by using the SAOS-based graphical user interface management system called AOUIMS [CHOI92]. The implementation of AOUIMS itself follows the SAOS approach, and even components

such as recorders can be implemented as SAOs. Although we have merged the graphical representation and behavior of a component in its class definition, it is possible to implement them separately.

4. A SAOS graphical editor allows us to construct a graphical representation of a SAOS program, from which the textual SAOS program can be generated. Furthermore, since SAOs are modularized well, it is possible to create a SAOS that allows reconfiguration while it is operating.

We consider that the SAOS approach is a new paradigm for programming object-oriented concurrent systems. It is especially useful for simulation systems.

References

[AGHA86] Agha, G. A. *Actors: A model of concurrent computation in distributed systems.* MIT Press, 1986.

[BIRT73] Birtwistle, G., Dahl, O. J., Byhrhang, B., and Nygard, K. *SIMULA BEGIN*, Auerbach, 1973.

[BLAC86] Black, A., Hutchinson, N., Jul, E., and Levy, H. Object structure in the Emerald system. Proc. Conf. on Object-Oriented Programming, 1986, pp. 78-86.

[BLOO87] Bloom, M. Mixed-mode simulators bridge the gap between analog and digital design, *Computer Design*, v26, January 15, 1987, pp51-63

[BOGA83] Bogard, T.F. *Computer Simulation of Linear Circuits and Systems*, John Wiley and Sons, 1983

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

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

[DAVE86] Dave, M. Mixed-mode Simulation on a PC-based workstation, *Electronics and Power*, v32, July 1986, pp523-526

[GOER88] Goering, R. A full range of solutions emerge to handle mixed-mode simulation, *Computer Design*, v27, February 1, 1988, pp 57-65

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

[KOHA88] Kohavi, Z. *Switching and Automata Theory*, McGraw-Hill, 1988

[LIPS90] Lipsett, R., Schaefer C., Ussery, C. *VHDL: Hardware Description and Design*, Kluwer Academic Publisher, 1990

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

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

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

[TAUB81] Taub, H., Schilling, D. *Digital Integrated Electronics*, McGraw-Hill, 1981

[YONE87] Yonezawa, A., Shibayama, E., Takada, T., and Honda, Y. Modeling and programming in an object oriented concurrent language ABCL/I. In *Object-Oriented Concurrent Programming*, Yonezawa, A. and Tokoro, M. (Eds), MIT press, 1987, pp. 55-90.

[ZISM78] Zisman, M. D. Use of production systems for modeling asynchronous, concurrent processes. In *Pattern-Directed Inference Systems*, Waterman, D.A. and Hayes-Roth, F. (Eds), Academic Press, 1978, pp. 53-69.