Practicing the Object Modeling Technology in a Functional Programming Framework

Jack B. Dennis and Handong Wu*
MIT Laboratory for Computer Science
Cambridge, MA, 02139, USA
e-mail: {dennis,handong}@abp.lcs.mit.edu
tel: 617-253-6837
fax: 617-253-6522

Object-oriented methodology has become an important approach to improving the quality of software and increasing the efficiency of its development. In this paper our aim is to show how the benefits of object-oriented methodologies can be realized within the framework of functional programming, achieving a more powerful level of modular program structure and avoiding introduction of the possibility of nondeterminate program behavior except when demanded by an application. We show how inheritance can be supported by an extension of the Sisal language, namely ObjectSisal, to include a natural class mechanism with immutable abstract data types, and how uses of "state" may be handled by means of stream data types and the nondeterminate merge operation. These principles are illustrated using examples from simulation and real-time systems. We believe that efficient distributed implementation of such an object-oriented methodology can be achieved through compile-time program transformation and execution platforms that implement fine-grain threads.
Keywords: object modeling technique, object-oriented programming, functional programming, nondeterminacy, distributed computing, real-time systems

1 Introduction

Object-oriented methodology has become an important approach to improving the quality of software and increasing the efficiency of its development. However, the evolution of distributed systems in the practice of software engineering has introduced chaos into object-oriented methods: When a typical object-oriented programming language is extended to support distributed applications, the semantic concepts based on abstract data types that have been useful for characterizing object models in sequential computation lose their validity, and reliance is palced on *ad hoc* operational models that fail to meet requirements

*On leave from Ericsson Telecom, Sweden

of encapsulation for the support of modular software construction. In particular, the acknowledged difficulties of programming using traditional primitives for coordinated multiprocessing reappear in most object-oriented languages that support concurrency.

Our objective in this paper is to show how the benefits of object-oriented methodologies can be realized in distributed computing within the framework of functional programming, achieving a more powerful form of modular program structure and avoiding introduction of the possibility of nondeterminate program behavior except when demanded by an application.

The presentation starts with some background observations regarding object-oriented methodology and problems in practicing object-oriented software engineering for a distributed or shared-memory multiprocessing computing environment. We take the *object modeling technique* (OMT) of Rumbaugh and his colleagues [24] as the point of departure for this discussion. In Section 3 we introduce *ObjectSisal*, a simple extension of the Sisal 2 functional programming language [21] to incorporate a *class* construct that supports user-defined immutable types, and we illustrate how concepts of object and inheritance are supported. We show how state machines can be implemented as stream-processing functions using the stream data types of Sisal, yielding functional, history-sensitive program modules. The requirement to support nondeterminate computation, in many real-time applications, for example, is met by introducing the *nondeterminate merge* operation [9] for data streams.

These ideas are illustrated by means of examples taken from real-world applications for which object-oriented methods are attractive: a logic simulation example in Section 4, and a telephone switching system in Section 5. In Section 6, we compare ObjectSisal with related programming languages and systems. We conclude in Section 7 with a summary and some remarks regarding the implementation of ObjectSisal.

2 Background

A usable approach to software engineering must cover the entire software life cycle from requirements analysis through design, implementation, maintenance, and extension. As with older approaches to software construction, object-oriented methods may be applied in all phases of software work. As often stated, one role of a programming language in object-oriented methodology is as a tool for carrying out the implementation phase. Of course, the closer the language is to reflecting concepts of the methodology, the easier it will be to implement a design.

The most fundamental concept in object-oriented software methodology seems to be its focus on data objects and their encapsulation together with the sets of operations to be used in connection with the role of the objects in the application program. This notion of *data abstraction* is in contrast to the emphasis on process and procedure in other software engineering approaches that appear to lack a structured way of dealing with data objects.

Although the concepts of abstract data types, which provide a precise formal foundation for the encapsulation of object classes, is a well-understood area in sequential programming languages [16], there is no agreement about how the abstraction should be carried over to parallel and distributed systems. The most common approach is simply to allow independent processes to have free use of access and update operations on arbitrary objects. This leads to the possibility of nondeterminate behavior of computations, whether or not repeatable behavior is desired by the programmer. Our view is that this is an unacceptable approach. Programmers should not have to expose themselves to unrepeatable behavior when that is not a requirement of the application!

There is nothing in the rationale for object-oriented software methodology, as stated by Blooch [6] and by Rumbaugh [24], that demands that objects have *mutable state*. Thus a real-world object that progresses through a significant sequence of states may be modeled by a *sequence* of state values, together with static data that describes fixed attributes of the object. Stream data types provide this capability, and our examples show how this approach may be practiced using ObjectSisal.

Inheritance is another aspect of object-oriented methodology that is held to be characteristic of the field. There seem to be two principal sorts of inheritance: The first corresponds to a subtype hierarchy generated by the "is-a" relation among object types. This form of inheritance may be added to a functional programming language by a simple extension of the syntax for defining abstract data types, as we illustrate in Section 3 for the Sisal language. The second object-oriented use of inheritance concerns aggregates, the "part-of" hierarchy. This concept can be applied simply by nesting (binding) type definitions as required by the structural relationships of data objects.

The motivation for inheritance appears to be the savings in program text and compiled code for operations that have similar effects for several specializations of a superclass of objects. This appears to be a valid argument where the operation does not apply to all subtypes of objects; when an operation applies to all subtypes (even if there are different effects for differing types of objects), it seems to be more efficient to apply the operation to the common superclass and use case analysis to determine the effect according to kind of object as given by an attribute of the superclass. These aspects of hierarchy will be illustrated in the examples of the next section.

We do not consider the topic of *multiple inheritance* because its semantics and utility are being questioned, even within the community of conventional objected-oriented advocates.

3 Principles

We have found the treatment of object-oriented methodology given by Rumbaugh and his colleagues [24] to be a solid exposition of the concepts and how they apply to real problems of software engineering. Rumbaugh considers three models for use in formulating the requirements of an application: the *object model*, the *dynamic model*, and the *functional model*. Here and in the examples given in later sections of this paper, we show how systems described in terms of these models may be implemented using ObjectSisal.

We have chosen Sisal 2 [21] as the functional programming language for expressing our examples because it includes stream data types that are important for our treatment

```
class Object is

  % Attributes

  attribute_1: type_1;
  attribute_2: type_2;
    -----
    -----

  % Operations

  function Operation_1
    -----
    -----
  end function Operation_1

  function Operation_2
    -----
    -----
  end function Operation_2

    -----
    -----

end class Object;
```

Figure 1: The `class` construct in ObjectSisal.

of state machines and nondeterminate computation. ObjectSisal includes extensions to provide for the encapsulation of sets of operations in *classes*, and to support inheritance through subtypes. The nondeterminate *merge* operation [9] is provided as the sole means for expressing nondeterminacy.

In ObjectSisal, an object model of the OMT takes the form of a static hierarchy of class definitions that define a root type and a collection of subtypes, in which each type or subtype is, formally, a set of immutable objects together with fixed operations, as explained below. A dynamic model of the OMT takes the form of a Sisal function (usually an operation of the class of the object supposed to have "state") that processes a stream of events to yield a stream of responses. This is illustrated by the phone line unit discussed in Section 5.

We limit our treatment of the OMT functional model to pure functions, which are invoked by class operations as required to handle events. Thus we omit treatment of *associations* of OMT and databases, which will be the subject of future work.

3.1 Objects, Classes, and Inheritance

In ObjectSisal a class of objects is denoted as shown in Figure 1. A class consists of a set of attributes and a set of operations. In consonance with principles of functional

```
function Create ( null returns Object )

  object Object [
    attribute_1: value_1;
    attribute_2: value_2 ]

end function Create
```

Figure 2: An operation of the `Object` class.

programming, a class represents an (immutable) *abstract data type*; each object of type `Object` has fixed attribute values.

Figure 2 shows a typical operation, in this case a `Create` operation that creates a new object of type `Object`. The keyword `object` begins a phrase having syntax similar to that of the Sisal record constructor; its value is an object of the class with specified attribute values. The keyword `private` may preceed `function` to indicate that the operation is only accessible to other operations of the class, and is invisible outside.

No means is provided for modifying an attribute value of an existing object. The consequence of insisting on this purity is that parallelism is exposed without needlessly introducing unrepeatable behavior, as our examples will illustrate.

Subtype declarations are permitted in ObjectSisal by an alternate header for a class:

```
class Object is SuperClass with
  -----
  -----
end class Object;
```

In this case, attributes and operations of the class `SuperClass` may be referenced by implementations of operations of the subclass `Object`. Of course, reuse of the names of attributes and operations in a subtype hierarchy is not permitted. A subtype is simply the immutable supertype, extended with additional data components and operations, and is itself a (immutable) derived abstract data type. (It is not a subtype in the sense of set membership, even though we may regard objects of the subtype as "elements" of the supertype class.) The immutable property of ObjectSisal objects avoids the reputed problems of "subtyping" in object-oriented languages.

A nice example of a subtype hierarchy in an application is the weather monitoring station example presented in [6], pages 293–326. The weather station reads data periodically from weather instruments, and keeps track of daily maximum and minimum values. It normally displays the current values reported by its sensors, but it also accepts interactive commands from a keypad that request display of historical (max/min) readings and support additional weather station functions.

The weather station has sensors for temperature, humidity, pressure, wind speed, and wind direction in the complete example. Here we omit humidity and wind speed for simplicity. The three remaining sensors generate the subtype hierarchy drawn in Figure 3

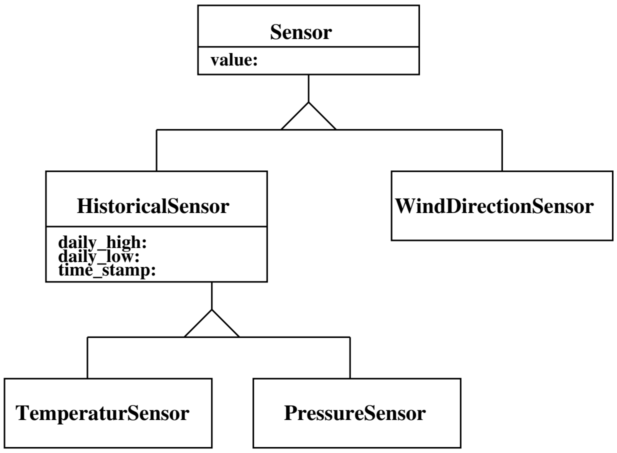

Figure 3: Portion of the object model for the weather monitoring station.

using the OMT diagramming notation. The class `HistoricalSensor` is a supertype of the `TemperatureSensor` and `PressureSensor` classes, and includes attributes for daily max and min values. It also has the operation `NewExtremes` which may be invoked by subtype operations to yield new `Sensor` instances with freshly calculated extreme values. The code to do this is illustrated in Figure 4.

3.2 Streams and State Machines

The Sisal language includes stream data types and the operations `stream_first` and `stream_rest` that yield the initial element of the given stream of values, and the stream consisting of the remaining elements, respectively. Figure 5 illustrates how a simple stream-processing computation may be expressed in Sisal as a tail-recursive function.

In [11] we have shown how such tail-recursive functions may be automatically transformed into iterative processes for efficient code construction. Several stream-processing modules may be combined to express a processing system in which pairs of modules are in producer/consumer relationships that permit all modules to be executing simultaneously [12].

This facility permits distributed simulation to be implemented in a programming style that does not allow non-repeatable (nondeterminate) behavior (to an observer). The logic simulation program discussed in Section 4 is an example.

A stream-processing function provides a natural style for expressing a dynamic model of the OMT. The set of states may be an enumeration type

```
type State = union [ q0, q1, ... qn: null ];
```

and the sequences of input and output events may be elements of data streams

```
class Sensor is
  value: real;
end class Sensor

class HistoricalSensor is Sensor with

  daily_high: real;
  daily_low:  real;
  time_stamp: integer;

  function NewExtremes (
    S: HistoricalSensor;
    time: integer;
    temp: Temperature;
    returns HistoricalSensor )
    let
      v := S.value;
      dh, dl := if NewDay (time, time_stamp)
        then v, v
        else (
         if v > S.daily_high then v else S.daily_high end if
         if v < S.daily_low then v else S.daily_low end if )
        end if
    in
      S [ daily_high: dh; daily_low: dl; time_stamp: time ]
    end let
  end function NewExtremes;
end class HistoricalSensor

class TemperatureSensor is HistoricalSensor with

  function SampleTemperature (
    TS: TemperatureSensor;
    time: integer;
    returns TemperatureSensor)
    let
      temp := ReadAirTemperature ();
      new_TS := NewExtremes  (TS [value: temp], time, temp);
    in
      new_TS
    end let
  end function SampleTemperature
end class TemperatureSensor
```

Figure 4: Sensor classes for the weather station example.

```
function StreamAdd (
   D: stream of integer;
   returns stream of integer )
   Accumulate ( D, 0 )
end function StreamAdd

function Accumulate (
   D: stream of integer;
   S: integer
   returns stream of integer )
   let
      sum := S + stream_first (D);
   in
      [ sum ] || Accumulate ( stream_rest (D), sum )
   end let
end function Accumulate
```

Figure 5: A simple stream-processing function written in Sisal.

```
type InEvent = union [ s0: in_info_0; .. sm: in_info_m ]
type InStream = stream of InEvent;
type OutEvent = union [ y0: info_0; .. ym: info_m ]
type OutStream = stream of OutEvent;
```

where the type of each alternative in the unions is chosen to represent the information conveyed by each kind of input or output event. Then the function representing the state machine may be written as shown in Figure 6. In this code the functions `TransitionFunction` and `OutputFunction` are the usual functions specifying state transitions and output symbols of a Mealy-type finite state machine. The coding of the phone line unit given in the appendix follows this scheme.

In some situations it is necessary to combine two stream-processing functions `f` and `g` as shown in Figure 7. This might be written as

```
let
   out_stream, y_stream := f (in_stream, x_stream );
   x_stream := g ( y_stream );
in
   out_stream
end let
```

However, this is not permitted in Sisal because the value name `x_stream` is used before it has been defined. This design decision was made so that Sisal could have *strict* semantics, that is, all arguments of any operation are completely evaluated before the operation may be applied. Because we wish a stream in ObjectSisal to have the properties of a *non-strict* list thereby permitting concurrent operation of producer and consumer modules, we include

```
function StateMachine (
     S: stream of InEvent;
     returns stream of OutEvent )

  MachineStep ( S, union State [ q0 ] )

end function StateMachine

function MachineStep (
     S: stream of InEvent;
     Q: State;
     returns stream of OutEvent )

  let
    first := stream_first ( S );
    rest := stream_rest ( S );

    new_q := TransitionFunction ( first, Q );
    new_y := OutputFunction ( first, Q );
  in
    stream [ new_y ] || MachineStep ( rest, new_q )
  end let

end function StateMachine
```

Figure 6: Expressing a dynamic (state-machine) model of the OMT in ObjectSisal.

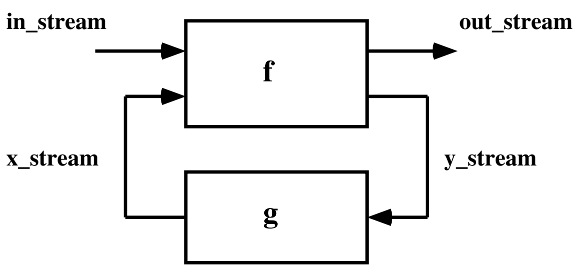

Figure 7: A combination of stream-processing functions.

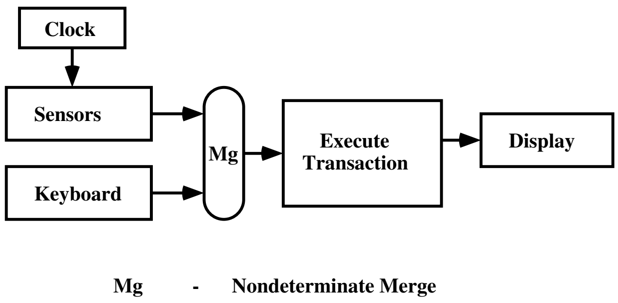

Figure 8: Flow of events in the weather monitoring station.

in ObjectSisal a feature that permits the value name of a stream argument to be introduced and referenced earlier in the program text than its “left-hand-side” appearance. Thus the above example may be written in ObjectSisal as

```
let
   out_stream, y_stream := f (in_stream, future_stream x_stream );
   x := g ( y_stream );
in
   out_stream
end let
```

In this, the key word `future_stream` asserts that the function argument `x_stream` denotes a stream of values and should be treated non-strictly. Illustrations of use of this feature occur in the weather station and telephone exchange examples that follow.

3.3 Nondeterminate Computation

While the logic simulation example has repeatable behavior and can benefit from being written using inherently repeatable programming methods, other computations cannot because they are inherently nondeterminate. The weather station with its asynchronous input from the keypad is an example. The local telephone switching system discussed in Section 5 is a more elaborate example.

The state of the weather station will be represented by a collection of attributes of a `WeatherStation` class in ObjectSisal. The set of attributes includes the set of `Sensor` objects. Two kinds of events call for two kinds of transactions to be performed by the weather station. A *sample transaction* is initiated by a clock tick and selected sensors are sampled according to the granularity of current time. A *keypad transaction* causes a temporary change in the display to present a user with certain derived measurement data, the dew point temperature, or max and min values, for example.

Figure 8 shows how the `Process` operation of the `WeatherStation` class is constructed using the merge operator to combine the two streams of transaction requests. The function `ExecuteTransaction` processes the resulting stream of requests. The ObjectSisal code corresponding to the diagram is given in Figure 9. This code would have repeatable (determinate) behavior except for its use of the `merge` operation.

In the telephone exchange we need a means of merging a number of streams unknown at compile time. To support this requirement in ObjectSisal, we extend the set of reduction operators that may be used in a Sisal `for` construct to include the merge operation. Thus the expression

```
for i in [1..n]
  S : stream of T := ... ;
returns merge of S[i]
end for
```

yields the stream resulting from merging the `n` streams defined in the body of the `for` loop.

These principles, and the examples that follow, show that the notion of "object with state" is not an essential element of an object-oriented software engineering methodology.

4 A Simulation Example

Synchronous logic simulation is a challenging problem for distributed computation. There is much potential parallelism, but it is difficult to exploit because the simulation process must behave as if a barrier synchronization were enforced between each pair of successive clock ticks. If this is actually done, then there is no possibility that any processing of later events may proceed while any work for the present clock tick remains incomplete. If, however, the action of any gate can be enabled by the availability of defined input logic levels, then no barrier synchronization is needed and much greater concurrency of simulation is possible (using a fine-grain parallel computer).

Synchronous logic simulation can be written in ObjectSisal in an elegant form. The logic simulation is performed by the `Simulate` operation of the `Circuit` class given in Figure 10. Two integer attributes tell the number of logic devices and the number of output signals of the circuit. The logic devices of the circuit are represented by an array of objects from the `Device` class (Figure 11) which is limited to "and" and "not" gates for simplicity. The network topology of the circuit is represented by objects in a `Connection` class (Figure 12) which are associated with each gate and map the inputs of each gate to outputs of other gates. The `out` array tells which gates produce the output signals of the circuit. The `Simulate` operation works by generating the sequence of `State` and `Output` arrays. Highly concurrent execution of the simulation process is possible if several generations of the state and output arrays are simultaneously live, and elements (logic values) are made accessible as soon as they become defined. For this example, we have assumed that a circuit exists, and have not shown any operations of the classes that would facilitate construction of circuits.

```
class WeatherStation is

  sensors: record [
    T_sensor: TemperatureSensor;
    P_sensor: PressureSensor;
    WD_sensor: WindDirectionSensor ];

  clock: Clock;

  keypad: KeyPad;

  function Create ( null returns WeatherStation )
    -----
    -----
  end function Create

  function Process ( WS: WeatherStation returns null )
    let
      command_stream := ParseInput (keypad);
      sample_stream :=
        GenerateSampleRequests (sensors, clock);
      transaction_stream :=
        merge (command_stream, sample_stream);
    in
      ProcessTransactions ( WS, transaction_stream )
    end let
  end function Process

  function ProcessTransactions (
      WS: WeatherStation;
      Ts: stream of Transaction;
      returns WeatherStation )
    let
      Tf := stream_first (tr_strm);
      WS_new := ExecuteTransaction ( WS, Tf )
    in
      ProcessTransactions ( WS_new, stream_rest (Ts) )
    end let
  end function ProcessTransactions

end class WeatherStation
```

Figure 9: The weather station program.

```
class Circuit is

  type State = array of boolean;
  type Output = array of boolean;

  node_cnt: integer;
  out_cnt:  integer;
  nodes: array of Device;
  net:   array of Connection;
  out:   array of integer;

  function Simulate ( C: Circuit; S: State;
      returns stream of State, stream of Output )

    let
      S_next := for device at i in C.nodes do
          s := Device.Operate (
             device,
             Connection.Select ( S, C.net [i]) );
        returns array of s;
        end for;

      O_next := for i in 1, out_cnt
        returns array of s [ C.out [i] ];
        end for;

      S_rest, O_rest := Simulate (C, S_next);

    in [ S_next ] || S_rest, [ O_next ] || O_rest

    end let

  end function Simulate;

end class Circuit;
```

Figure 10: The simulation process written as a `Circuit` class in ObjectSisal.

```
class Device is

  kind: union [ AndGate, NotGate: null ];

  function Operate ( D: Device; L: array of boolean;
      returns boolean )

    case Device.kind of
      AndGate: L [1] & L [2];
      NotGate: ~L [1];
    end case

  end function Operate;
end class Device;
```

Figure 11: The `Device` class for the logic simulation example.

```
class Connection is

  input_cnt: integer;
  links: array of integer;

  function Select ( C: Connection; L: array of boolean;
      returns boolean )

    for i in 1, C.input_cnt
    returns array of L [ C.links [i] ]
    end for

  end function;

end class Connection;
```

Figure 12: The `Connection` class for the logic simulation example.

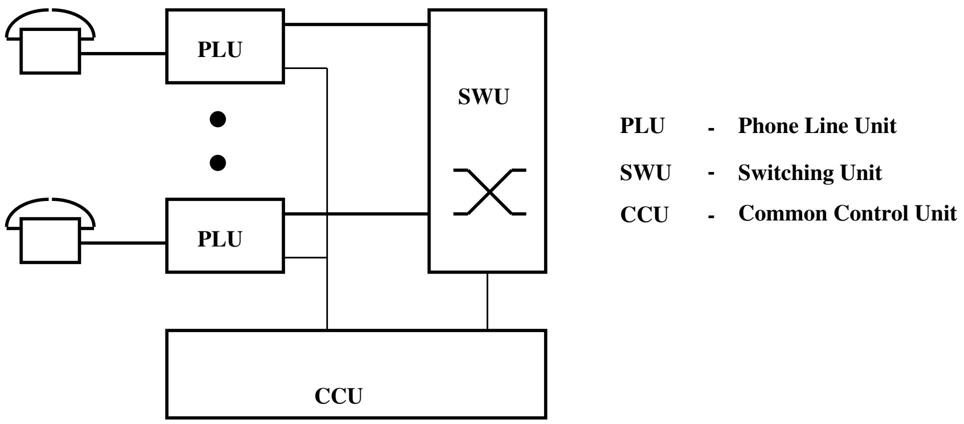

Figure 13: Generic structure of a telephone switching exchange.

5 Telephone Exchange

The simple telephone exchange allows any subscriber to request that a speech connection be set up to any other subscriber. As shown in Figure 13 the exchange has three kinds of components: phone line units (PLUs), the switch unit (SWU), and the common control unit (CCU). For simplicity, we assume that each phone line unit handles just one subscriber. The state of a PLU notes whether the phone's handset is `off_hook` or `on_hook`, and also makes the phone sound signals: `ring`, `busy_tone`, `ring_tone`. The switch unit provides circuits for connecting any pair of subscribers. The common control unit receives request and status messages from the other units and issues control messages to them. Here and in the appendix we present an executable specification of the telephone exchange, written as an `Exchange` class in ObjectSisal. Figure 14 shows the relations among the objects of an instance of the `Exchange` class. The components interact via several event streams conveyed between pairs of objects. The elements of these streams have type definitions in ObjectSisal as given in Figure 15; the names are chosen from the viewpoint of the common control unit.

Figures 16 and 17 present the `Exchange` class which amounts to a formal (but nevertheless executable) specification of the interconnections shown in Figure 14. It is the top level of the telephone exchange implementation. Its `Create` operation builds an `Exchange` object containing a SWU, a CCU, and the specified number of PLUs. In the `Run` operation, the streams of events from the PLUs and the SWU are merged into a single input stream for the `Process` function of the `CCU` object. Because the interconnection of units is cyclic, the `future_stream` keyword is used to declare references to stream values yet to be defined.

The phone line unit has a significant dynamic model represented by the state diagram in Figure 18. The states are used for noting the progress of making a connection on behalf of a calling subscriber, acknowledging the off-hook response of the called subscriber, and clearing the connection on completion of the call (on-hook condition of either subscriber). The ObjectSisal implementation of the PLU given in the appendix merges the streams of ex-

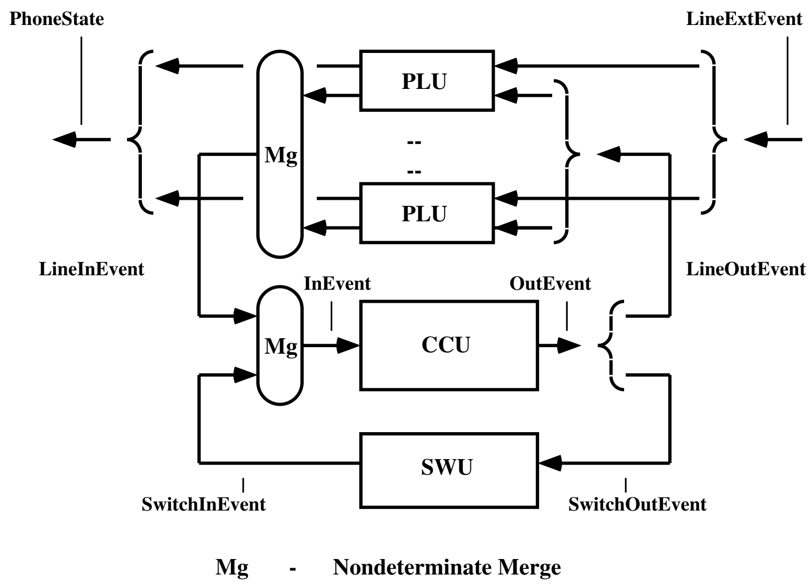

Figure 14: Inter-object streams in the local telephone exchange.

ternal and command events, and uses a tail-recursive state-transition function `ProcessStep` exactly as discussed in Section 3.

The CCU responds to signals from the PLUs by asking the SWU to make and clear paths in the switching network of the exchange, and the SWU maintains the connections and informs the CCU of its success or failure to create a path. The details of the ObjectSisal specifications of these units are given in the appendix.

6 Related Work

The design of ObjectSisal has roots in earlier research efforts, including Clu's abstract data type mechanism [16], Dennis and Weng's work on tail-recusive computing with streams [8, 10], proposals for structuring nondeterminate computations [9], and the common understandings of inheritance and encapsulation as these notions have evolved in object-oriented programming paradigms.

Here we compare ObjectSisal with other advanced languages and systems devised for programming distributed systems, both object-oriented and others that offer the benefits of high-level structured programming in real-time applications. Our discussion considers how well each programming environment supports principles of modular software construction.[1]

[1]Modularity mechanisms for concurrent, object-oriented programming are discussed in [2], but no definition of modularity is given and the sort of program modularity described apprears to be quite limited. For example, no properties corresponding to our Recursive Construction and Secure Arguments principles are

```
% Subscriber phone numbers

type Number = integer;

% External input events to the PLUs

type LineExtEvent =
   union [ off_hook, on_hook: null; digits: integer ];

% Status Signals from the PLUs

type PhoneState = union [
   idle, dialing, connect, busy_tone,
   ring_tone, ring, speech: null];

% Input events for CCU from the PLUs

type LineInEvent = union [
   seize: record [called: Number];
   clear, answer: null ];

% Input events for CCU from the SWU

type SwitchInEvent = union [
   success: record[ calling, called: Number ];
   failure: record[ calling: Number ] ];

% Output events for PLUs from the CCU

type LineOutEvent =
   union [ sound_ring_tone, start_ring, sound_busy_tone, stop_ring: null ];

% Output events for the SWU from the CCU

type SwitchOutEvent = union [
   connect: record[ calling, called: Number ];
   disconnect: record[ line: Number ] ];
```

Figure 15: Global event types for the telephone exchange.

```
class Exchange is

  line_cnt: integer;
  line_units: array of PLU;
  switch_unit: SWU;
  control_unit: CCU;

  function Create (n: integer returns Exchange )
    let
      swu := SWU.Create (n);
      ccu := CCU.Create (n);
      plus := for line in [1..n] do
        returns array of PLU.Create ()
        end for;
    in
      object Exchange [line_cnt: n;
        line_units: plus; switch_unit: swu; control_unit: ccu ]
    end let
  end function Create;

  function Run (
      exchange: Exchange;
      ext_events: array of stream of LineExtEvent;
      returns array of stream of PhoneState )
  end function Run
end class Exchange
```

Figure 16: The class `Exchange` specifying a simple telephone exchange.

```
function Run (
    exchange: Exchange;
    ext_events: array of stream of LineExtEvent;
    returns array of stream of PhoneState )
  let
    plu_status_streams, plu_in_streams :=
      for line in [1..line_cnt] do
         line_unit := line_units[line];
         status_stream, in_stream := line_unit.Process (
            ext_events [line],
            stream  plu_out_streams [line] );
      returns
        array of status_stream,
        array of in_stream
      end for;
    merged_plu_inputs := for line in [1..line_cnt] do
      returns merge of TagStream (line, plu_in_streams [line])
      end for;
    swu_in_stream := switch_unit.Process (future_stream swu_out_stream);
    in_stream := merge ( merged_plu_inputs, swu_in_stream);
    plu_out_streams, swu_out_stream := control_unit.Process (in_stream);
  in plu_status_streams
  end let
end function Run
```

Figure 17: The `Run` operation of the `Exchange` class.

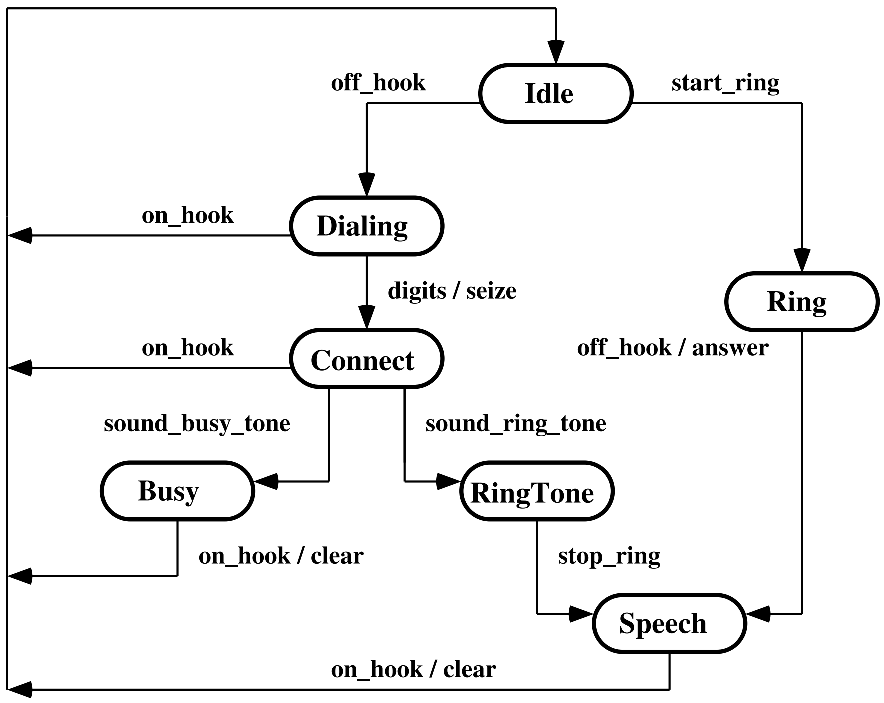

Figure 18: Dynamic model (state diagram) for the phone line unit.

In [10] we listed six principles of modular software construction:

1. *Information Hiding Principle:* The user of a module must not need to know anything about the internal mechanism of the module to make effective use of it.
2. *Invariant Behavior Principle:* The functional behavior of a module must be independent of the site and context from which it is invoked.
3. *Data Generality Principle:* The interface to a module must be capable of passing any data object an application may require.
4. *Secure Arguments Principle:* A module must not be able to affect the behavior of another module through a shared argument.
5. *Recursive Construction Principle:* A program constructed from modules must be usable as a component in building larger programs or modules.
6. *System Resource Management Principle:* Resource management for data objects must be performed by the computer system, not by program modules.

We believe that the ObjectSisal Language provides a programming framework that satisfies all six of these principles. Most other approaches to distributed object-oriented computation do not.

Some tools for concurrent object-oriented programming make explicit use of sequential processes. The Ada language uses *tasking* to spawn concurrent processes that can operate on objects using methods encaspsulated in *packages*. ConcurrentSmalltalk [15] uses processes so that the concurrent version can be made compatible with its sequential parent, Smalltalk 80. In the "composable" lanaguage CC++ [7], C++ is augmented with processes and a synchronizing mechanism to coordinate operations on objects.

Orca [5], is intended to support concurrent applications on a network of machines that does not support a common address space. An Orca program consists of processes that execute operations on objects. By using relatively coarse grain operations, operations on objects at remote sites can be executed efficiently in comparison to the fine-grain read and write primitives used in other approaches to distributed computing. However, nested objects (objects that are components of other objects), lead to inefficiencies due to the complexity of ensuring atomicity of operations.

These tools all view objects in the traditional object-oriented manner—as entities having "state"—which leads to all of the difficulties concerning coordination of actions and conflicts with principles of modularity discussed above. They fail to satisfy the principles Data Generality and Recursive Construction, and most of them also fail to support the Resource Management Principle. Data Generality is violated because it is normally not possible to pass an arbitrary data value (object) in a message sent from one process to another. (A general mechanism must be able to pass addresses of component objects, and these addresses

discussed.

must have the same meaning in the context of any process, a property not usually satisfied.) The principle of Recursive Program Construction is usually violated because there is no syntactic construct that permits encapsulation of a set of processes to present the same form of interface to users as the class construct provides for object types. The Resource Management principle is violated when processes are statically bound to processors, and cannot be migrated by the system to balance resource usage; memory management is limited by the binding of allocation actions to processes.

Several approaches to distributed object-oriented programming do not have the problems of explicit processes. The most studied and developed of these are the actor-based paradigms developed by Agha and Hewitt [1], the work of Yonezawa and his assocites on ABCL [29], and others, for example POOL-T [3]. In these systems, a computation is performed by a collection of *actors* that respond to *messages* containing data (argument values and objects) and the name of a *method* to be executed on the object. Sending a message corresponds to performing a function or procedure call in a conventional language, but the method defines an operation characteristic of the object, and is encapsulated with object data in a *class* definition.

Because the methods of a class may alter values of object attributes, undisciplined behavior would occur if concurrent execution of methods for the same object were permitted. Therefore, actor languages provide synchronization mechanisms that may be used to ensure that the effect of operations is as if they have been strictly interleaved. Many different schemes have been described in the literature; however they all have the property of permitting accesses and updates of objects to be arbitrarily interleaved, therby permitting nondeterminate (nonrepeatable) behavior. Because actor languages permit messages to contain objects that may be updated by methods, they also do not satisfy the Secure Arguments Principle.

The actor-based systems generally satisfy Data Generality and Resource Mangement principles of software modularity. They satisfy the Data Generality principle by permitting any object to be sent in a message, and the Resource Mangement Principle through use of system memory management (including automatic garbage collection). They also support the Information Hiding principle through the class encapsulation facility.

Support of the Recursive Construction Principle demands a means for combining modules (class definitions) into new modules that can be reused in different contexts with invariant behavior. This principle is supported by mechanisms for importing class definitions into other classes and for binding methods to operation calls. Dynamic binding that supports type polymorphism is good; however, dynamic binding mechanisms that lead to context-dependent behavior are in violation of the Invariant Behavior principle. In Object-Sisal, static binding of methods (functions) to function invocations is employed, so recursive construction is supported with a guarantee of invariant behavior.

UFO (United Functions and Objects) [27] is perhaps the best-known work attempting to combine the features of object-oriented and functional programming styles. Besides a purely functional subset, UFO supports "stateful" objects in the same way as other concurrent object-oriented languages, supporting the view that objects may be updated by concurrent operations. Calls upon the functions and procedures of a class of stateful objects are handled as messages, using mechanisms similar to those of the actor-based languages.

It is asserted in [27] that functions in UFO are not permitted to modify their arguments, as demanded by our Secure Arguments Principle. It appears that this is to be implemented by compile-time determination of whether any argument to a function can contain a stateful object to which a procedural method could be applied. In ObjectSisal we expect the Secure Arguments Principle to be met because most data are values. However, it is conceivable that use of the nondeterminate merge operator can lead to the creation of data objects that can exhibit changing behavior. We must be certain that such data objects cannot be made components of function arguments, but we have not yet done a careful study of this question.

Although UFO and ObjectSisal share the common motivation of realizing the benefits of the functional programming style in distributed computing applications, two very different approaches to the problem of state are used. In using stateful objects, we believe UFO imports weaknesses of concurrent object-oriented programming, including, in particular, the property that programs that utilize "objects" are susceptible to nondeterminate behavior, whether or not intended.

In ObjectSisal, we insist that all objects are immutable and that those situations that appear to require "state" can be supported using history-sensitive functions on data streams to build state machines, and by using the merge of streams when nondeterminate behavior is required. One property of this design is that ObjectSisal programs cannot deadlock except in case of memory exhaustion, a property not achieved by other programming tools for object-oriented software engineering.

In [28], Shapiro and Takeuchi have shown how object-oriented programming may be done using Concurrent Prolog. Shared logic variables are used to implement streams of events flowing between objects, which are processes executing self-recursive functions. Because Concurrent Prolog allows nondeterminacy in the way subgoals are selected, programs written in Concurrent Prolog are subject to the same hazards as in other object-oriented styles. Indeed, the nondeterminate merge has a simple implementation in Concurrent Prolog.

Many object-oriented languages are typeless or rely on runtime detection of some kinds of type errors. ObjectSisal does not utilize dynamic binding, so every function interface can be fully type-checked before program execution; no type errors should show up during program execution. Sisal types are sets of immutable data objects with operations forming abstract heterogeneous algebras; hence the type system is not burdened with duties relating to protection or concurrency control, as is done in some object-oriented languages.

Before leaving object-oriented programming systems, we note that there is an acknowledged interference between the goals of inheritance (making the methods of one class available to other classes that define subtypes) and mechanisms designed to support coordination between access and update operations. A thorough analysis of this problem has been done by Matsuoka and Yonezawa [19], yielding no completely satisfactory solution. As a consequence, the designers of many concurrent object-oriented languages have elected to forgo implementing inheritance. In ObjectSisal the problem may not arise because "messages" (function calls in ObjectSisal) are never selected or queued. The Maude language [20] also appears to avoid the anomaly by adopting a "declarative" semantics based on a formal rewriting logic.

Erlang [4] is a "declarative" programming langauge that has been used experimentally to implement a prototype telephone switching system [23]. In a distributed context, an Erlang program creates explicit processes, and uses message passing for interprocess communication. The use and benefit of functional programming lies in the construction of the individual proocess code. There appears to be no means in Erlang for encapsulating a group of processes into a higher level program module.

These features of Erlang are at odds with the principles of Data Generality, Secure Arguments, Recursive Construction, and Resource Management. If a system to be built is amenable to static resource management, that is, compile-time partitioning and scheduling is feasible, then Erlang provides support for writing the application as a single layer of concurrent modules that coordinate through message-passing. However there is no way to group sets of these modules into larger system components that may be used similarly. Furthremore, Data Generality is limited by the data types that may be packed into messages.

In this paper we have not addressed the question of applying functional programming principles and software modularity principles to the issues of database systems. Most object-oriented database systems are based on a transaction-oriented view [22]. A particularly sophisticated design is Thor [17]. As in actor-based object-oriented languages, consistency of the set of stored objects is maintained by imposing mutual exclusion on calls to methods of object classes. This is done either by some atomicity scheme using two-phase commit, or by means of locks. Thor supports applications written in Theta [18], a sequential object-oriented language evolved from Clu.

It seems that ObjectSisal would work well for object-oriented databases. An implementation would treat all objects as persistent and use a garbage collector to reclaim memory of abandoned objects. The "associations" of the Rumbaugh OMT method can be built using standard classes (of immutable objects); "stores" may be realized as state machines, if determinate, and using the nondeterminate merge of transaction streams otherwise.

7 Conclusion

In this paper we have demonstrated, using several programming examples adapted from real-world applications, that significant benefits of object-oriented software methodology can be achieved through simple extension of a purely functional programming language. Our approach using ObjectSisal also provides a more effective way of addressing issues arising in programming for parallel and distributed systems. In ObjectSisal, parallelism is exposed naturally and without introduction of mechanisms that interfere with support for the principles of modular software construction. Objects having "state" are represented using functions that process streams of values, and *nondeterminate* behavior is limited to programs that use the nondeterminate merge operation to assign an order to transaction requests.

We have noted in the course of this presentation that the full benefit of using ObjectSisal in software engineering depends on the exploitation of fine-grain concurrency by a target distributed computing system. The characteristics of such *multithreaded* computer systems have been discussed by Dennis and Gao [13, 14] and a process for static analysis and mapping of stream-based programs has been discussed in [12].

References

[1] G. Agha. Concurrent object-oriented programming. *Communications of the ACM 33*, 9 (September 1990), pages 125–141.

[2] Gul Agha, Svend Frolund, Woo Y. Kim, Rajendra Panwar, Anna Patterson, and Daniel Sturman. Abstraction and modularity mechanisms for concurrent computing. In *Research Directions in Concurrent Object-Oriented Programming*, Agha, Wegner, and Yonezawa, Eds., Cambridge: The MIT Press, 1993. Chapter 1, pp 3–21.

[3] Pierre America. POOL-T: A parallel object-oriented language. In *Object-Oriented Concurrent Programming*, Yonezawa and Tokoro, Eds., Cambridge: The MIT Press, 1987. Pages 199–220.

[4] J. L. Armstrong and S. R. Virding. Erlang: An experimental telephony programming language. In Proceedings of the International Switching Symposium, Stockholm, 1990

[5] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tenenbaum. *Orca: A language for parallel programming of distributed systems. IEEE Trans on Software Engineering*, 18(3): 190–205 (March 1992).

[6] Grady Blooch. *Object-Oriented Analysis and Design with Applications.* Second Edition, Benjamin/Cummings, 1994.

[7] K. Mani Chandy and Carl Kesselman. CC++: A declarative concurrent object-oriented programming notation. In *Research Directions in Concurrent Object-Oriented Programming*, Agha, Wegner, and Yonezawa, Eds., Cambridge: The MIT Press, 1993. Chapter 11, pp 281–313.

[8] J. B. Dennis and K. S. Weng. An abstract implementation for concurrent computations with streams. In *Proceedings of the 1979 International Conference on Parallel Processing*, pages 35–45, 1979.

[9] Jack B. Dennis. A language design for structured concurrency. In *Design and Implementation of Programming Languages.* In *Lecture Notes in Computer Science*, No. 54, pages 231–242. Springer-Verlag, 1977.

[10] Jack B. Dennis. Machines and models for parallel computing. *International Journal of Parallel Programming*, Vol. 22, No. 1, pages 47–77, February 1994.

[11] Jack B. Dennis. Stream Data Types for Signal Processing. In *Advances in Dataflow Architecture and Multithreading*, J.-L. Gaudiot and L. Bic, editors. IEEE Computer Society Press, 1995.

[12] Jack B. Dennis. Static mapping of functional programs: An example in signal processing. *Proceedings of the Conference on High Performance Functional Computing*, pp 149–163. Livermore, CA: Lawrence Livermore National Laboratory, April 1995.

[13] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: principles, projects, and issues. In Robert A. Ianucci, editor, *Advances in Multithreaded Computer Architecture*. Kluwer, 1994.

[14] Jack B. Dennis and Guang R. Gao. Multiprocessor Implementation of Nondeterminate Computations in a functional programming framework. Computation Structures Group Memo 375, Laboratory for Computer Science, MIT, 1995.

[15] Yasuhiko Ishikawa and Mario Tokoro. Concurrent programming in ConcurrentSmalltalk. In *Object-Oriented Concurrent Programming*, Yonezawa and Tokoro, Eds., Cambridge: The MIT Press, 1987. Pages 129–158.

[16] Barbara Liskov. Abstraction mechanisms in CLU. *Communications of the ACM 20*, 8 (August 1977), pages 564–576.

[17] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In *Distributed Object Management*, Morgan-Kaufmann, 1994. Pges 79–91.

[18] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawat, Robert Gruber, Paul Johnson, and Andrew C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, 1995.

[19] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, Agha, Wegner, and Yonezawa, Eds., Cambridge: The MIT Press, 1993. Chapter 4, pp 107–150.

[20] Jose Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In *Research Directions in Concurrent Object-Oriented Programming*, Agha, Wegner, and Yonezawa, Eds., Cambridge: The MIT Press, 1993. Chapter 12, pp 314–390.

[21] R. R. Oldehoeft, A. P. W. Bohm, D. C. Cann, and John T. Feo. SISAL Reference Manual: Language Version 2.0. Technical report, Lawrence Livermore National Laboratory and Colorado State University, 1992.

[22] M. Tamer Ozsu, Umeshwar Dayal, and Patrick Valduriez. An introduction to distributed object management. In *Distributed Object Management*, Morgan-Kaufmann, 1994. Pges 1–24.

[23] M. Persson, K. Odling, and D. Eriksson. A switching software architecture prototype using real time declarative language. In *Proceedings of the International Switching Symposium*, volume 2, page C7.3. Institute of Electronics, Information, and Communication Engineers, Japan, 1992.

[24] James Rumbaugh, et al. Object-Oriented Modeling and Design. Englewood Cliffs: Prentice Hall, 1991,

[25] John Sargeant. *Uniting functional and object-oriented programming.* In *Object Technologies for Advanced Software. Lecture Notes in Computer Science*, No. 742, Springer-Verlag, 1993. Pages 1–26.

[26] John Sargeant, Chris Kirkham, and Steve Anderson. Towards a computational Model for UFO. In *Parallel Architectures and Compilation Techniques*, In *IFIP Transactions*, Volume A-50. Amstetdam: North-Holland, 1994. Pages 299–308.

[27] John Sargeant. *United Functions and Objects: An Overview.* Technical Report UMCS-93-1-4, Manchester University, 1993.

[28] Ehud Shapiro and Akikazu Takeuchi. Object oriented programming in Concurrent Prolog. In *Concurrent Prolog: Collected Papers*, Volume 2, Chapter 29, pp251–273. Cambridge: MIT Press, 1987.

[29] Akinori Yonezawa, Ed. *ABCL: An Object-Oriented Concurrent System.* Cambridge: The MIT Press, 1990.

Appendix: ObjectSisal Specifications

In this appendix we give ObjectSisal programs that are executable specifications of the three object modules of the illustrative telephone exchange. The top-level specification has already been given as the `Exchange` class in Figure 16. The program for each component module is a class definition that includes a tail-recursive `Process` operation that acts on successive elements (events) of an input stream. In each case, the `Process` operation may be regarded as a state machine; it is a trivial state machine (one state) in the cases of the CCU and the SWU, where the action taken is determined by the next input event and is selected by a simple `case` expression. In the case of the PLU a two-level `case` expression is used because the action taken is determined by the combination of phone line state and input event. We have extended Sisal here by allowing the tags of a nested union type to be combined by a dot, thereby avoiding a nested case expression. Each class has a `Create` operation for use in creating an instance (object) of its principal type.

The common control unit is specified by the `CCU` class in Figure 19. Its `ProcessStep` operation (Figures 20 and 21) generates the streams of command events sent to the SWU and the response signals to the PLUs. A special private function `EnterLineEvent` (Figure 22) is used to construct an array of prefix streams (most of them empty) containing any output events to be sent to PLUs. It serves as a *distributor* of output events to the various PLUs. It is a pure function, as are all operations of the three classes defined here, with the exception of the `Process` operation of the `PLU` class which combines input streams using the nondeterminate merge operator.

The phone line unit is specified by the `PLU` class shown in Figure 23. As in the `Exchange` class, the two input streams are merged and presented to the `ProcessStep` function with the initial state `Idle`. The coding of the `ProcessStep` function (Figures 24 and 25) is straightforward. The specification of the switch unit, given in Figure 27, involves no new features. However, we have not indicated a representation for the SWU state, but use the unelaborated functions `FreePath`, `AddPath`, and `DeletePath` to test and modify connections made through the switch.

For completeness we also give (Figure 28) the coding of the `TagStream` function used in the `Run` operation of the `Exchange` class.

```
class CCU is

  line_cnt: integer;

  function Create (n: integer returns CCU )

    object CCU [line_cnt: n]

  end function Create;

  function Process (
    in_stream: stream of InEvent;
    returns
      array of stream of LineOutEvent,
      stream of SwitchOutEvent )

    let origin := array_fill (1, line_cnt, 0);
    in
      ProcessStep ( origin, in_stream )
    end let
  end function Process

  private function ProcessStep (
    origin: array of Number;
    in_stream: stream of InEvent;
    returns
      array of stream of LineOutEvent,
      stream of SwitchOutEvent )

    ------
    ------

  private function EnterLineEvent

    ------
    ------

end class CCU
```

Figure 19: The class `CCU`, specifying the common control unit.

```
private function ProcessStep (
  origin: array of Number;
  in_stream: stream of InEvent;
  returns
    array of stream of LineOutEvent,
    stream of SwitchOutEvent )
  let
    in_tail := stream_rest (in_stream);
    in_event := stream_first (in_stream);
    new_origin, plu_events, swu_events :=
      case in_event of
        line:
          let
            line := in_event.tag;
            event := in_event.event;
          in case event of
              seize:
                origin,
                stream [],
                stream [union SwitchOutEvent [
                  connect: record [
                    calling: line;
                    called: event.called ] ];
              answer:
                let
                  calling_line := origin [line];
                  line_event := union LineOutEvent [
                    command.stop_ring ];
                in
                  origin,
                  EnterLineEvent (plu_events, calling_line, line_event),
                  stream [union SwitchOutEvent [
                    connect: record [
                      calling: line;
                      called: event.called ] ]
                end let;
              clear:
                origin [line: 0],
                stream [],
                stream [union SwitchOutEvent [
                  disconnect: record [line: line] ];
            end case
          end let;
```

Figure 20: Specification of the Common Control Unit. Part 1: Line events.

```
        switch.success:
          let
            calling_line := in_event.calling;
            called_line := in_event.called;
            line_event_a := union LineOutEvent
              [command.ring_tone];
            line_event_b := union LineOutEvent
              [command.ring];
          in
            origin [called_line: calling_line],
            EnterLineEvent (
              EnterLineEvent (
                plu_events, calling_line, line_event_a),
                called_line, line_event_b),
            stream []
          end let;
        switch.fail:
          let
            calling_line := in_event.calling;
            line_event_a := union LineOutEvent [
              command.busy_tone];
          in
            origin,
            EnterLineEvent (plu_tail, A, line_event_a),
            stream []
          end let;
      end case
    plu_tail, swu_tail := Process (new_origin, in_tail);
    plu_out_streams :=
      for line in 1..line_cnt do
        plu_out_stream := plu_events [line] || plu_tail [line];
      returns array of plu_out_stream
      end for
    swu_out_stream := swu_events || swu_tail;
  in
    plu_out_streams,
    swu_out_stream
  end let
end function ProcessStep
```

Figure 21: Specification of the Common Control Unit. Part 2: Switch events.

```
private function EnterLineEvent (
    plu_events: array of stream of LineOutEvent;
    line: Number;
    event: LineOutEvent;
    returns array of stream of LineOutEvent )
  for i in [1..line_cnt] do
    line_events :=
      if i = line
      then stream [ event ] || plu_events [i]
      else plu_events [i]
      end if;
  returns array of line_events
  end for;
end function EnterLineEvent
```

Figure 22: The auxiliary function `EnterLineEvent` for the `CCU` class.

```
class PLU is

  type LineEvent = union [
    external: LineExtEvent;
    command: LineOutEvent ];

  type LineState = union [
    Idle, Dialing, Connect, RingTone, BusyTone, Ring, Speech];

  function Create ( returns PLU )
    object PLU
  end function Create;

  function Process (
    Ex: stream of LineExtEvent;
    Eo: stream of LineOutEvent;
    returns
        stream of PhoneState,
        stream of LineInEvent )
    let
      El := merge ( Ex, Eo );
      S0 := union LineState [Idle];
    in  ProcessStep (El, S0)
    end let
  end function Process;

  private function ProcessStep
      ------
      ------
  end function ProcessStep

end class PLU
```

Figure 23: The class `PLU`, specifying the phone line unit.

```
private function ProcessStep (
    in_stream: stream of LineEvent;
    S:  LineUnitState;
    returns
        stream of LineInEvent,
        stream of PhoneState )
  let
    in_event := stream_first ( in_stream );
    in_tail := stream_rest ( in_stream );
    out_flag, out_event, status, new_state :=
      case S of
        Idle: case in_event of
            external.off_hook:
              false, record [],
              union PhoneState [dialing],
              union LineState [Dialing];
            command.ring:
              false, record [],
              union PhoneState [ring],
              union LineState [Ring];
            otherwise:
              false, record [],
              union PhoneState [idle] ),
              union LineState [Idle] );
          end case
        Dialing: case in_event of
            external.on_hook:
              false, record [],
              union PhoneState [idle],
              union LineState [Idle];
            external.digits:
              let
                dialed := digits;
              in true,
                union LineInEvent [seize: record [
                  called: dialed] ],
                union PhoneState [connect],
                union LineState [Connect]
              end let;
            otherwise: false, record [],
              union PhoneState [dialing]
              union LineState [Dialing];
          end case
```

Figure 24: State transition function of the phone line unit. Part 1.

```
Ring: case in_event of
    external.off_hook: true,
      union LineInEvent [answer],
      union PhoneState [speech],
      union LineState [Speech];
    otherwise: false, record [],
       union PhoneState [ring];
       union LineState [Ring];
  end case
Connect: case in_event of
    external.on_hook: true,
      union LineInEvent [clear],
      union PhoneState [idle],
      union LineState [Idle];
    command.ring_tone: false, record [],
      union PhoneState [ring_tone],
      union LineState [RingTone];
    command.busy_tone: false, record [],
      union PhoneState [busy_tone],
      union LineState [Busy];
    otherwise: false, record [],
      union PhoneState [connect],
      union LineState [Connect];
  end case
```

Figure 25: State transition function of the phone line unit. Part 2.

```
    RingTone: case in_event of
        external.on_hook: true,
          union LineInEvent [ clear: record [calling: self]],
          union PhoneState [idle],
          union LineState [Idle];
        command.stop_ring: false, record [],
          union PhoneState [speech],
          union LineState [Speech];
        otherwise:
          false, record [],
          union PhoneState [ring_tone],
          union LineState [RingTone];
      end case
    Busy: case in_event of
        external.on_hook: true,
          union LineInEvent [ clear: record [calling: self] ],
          union PhoneState [idle],
          union LineState [Idle];
        otherwise: false, record [],
          union LineUnitState [busy];
      end case
    Speech: case in_event of
        external.on_hook: true,
          union LineInEvent [clear: record [calling: self] ],
          union PhoneState [idle],
          union LineState [Idle];
        otherwise: false, record [],
          union PhoneState [speech],
          union LineState [Speech];
      end case
  end case

  out_tail, status_tail := Process ( in_tail, new_state );
 in if out_flag then
    stream [out_event] || out_tail,
    stream [status] || status_tail,
  else
    out_tail,
    stream [status] || status_tail,
  end if
 end let

end function ProcessStep
```

Figure 26: Transition function for the phone line unit. Part 3.

```
class SWU is

  type SwitchState = ... ;

  function Create (state0: State;
    returns SWU )
    object SWU [];
  end function Create;

  function Process (
      in_stream: stream of SwitchOutEvent;
      returns stream of SwitchInEvent )
    ProcessStep (in_stream, switch_initial_state);
  end function Process;

  prviate function ProcessStep (
    in_stream: stream of SwitchOutEvent;
    state: SwitchState
    returns stream of SwitchInEvent )
    let
      in_event := stream_first (in_stream);
      in_tail := stream_rest (in_stream);
      switch_events, new_state :=
       case event of
         seize: if FreePath ( state, in_event.calling, in_event.called )
           then
             stream [ union SwitchInEvent [ success: record [
               calling: in_event.calling;
               called: in_event.called ] ] ],
             AddPath ( state, in_event.calling, in_event.called )
           else stream [ union SwitchInEvent [ fail: record [
               calling: in_event.calling ] ] ],
             state
           end if
         clear: stream [],
           DeletePath ( state, in_event.line );
       end case
    in
      switch_events || ProcessStep (in_tail, new_state)
    end let
  end function ProcessStep
end class SWU
```

Figure 27: Executable specification of the switch unit.

```
private function TagStream (
  line: integer;
  strm: stream of LineEvent;
  returns stream of TaggedLine Event )

  stream [ record [
      tag: line;
      event: stream_first (strm) ] ||
    TagStream (line, stream_rest (strm)
end function TagStream
```

Figure 28: The `TagStream` function for tagging phone line events with the line number.