Systems Modeling, Simulation and Analysis  
Using COVERS Active Objects  

Andrei V. Borshchev, Yuri G. Karpov and Victor V. Roudakov  
St.Petersburg Technical University, Russia  
covers@dcn.nord.nw.ru  http://dcn.nord.nw.ru

Abstract
We consider a modeling language and a simulation environment based on object-oriented principles and aimed to help in the design of reactive systems. The language framework includes diagrams of object structure and interconnection, statecharts as a behavior description, and C++ for data objects and functions. Timed Transition Systems are used as a semantical model. COVERS enables the user to perform the whole modeling-simulation-analysis cycle within a single MS Windows-based graphical environment. We overview the code generation, model execution and visualization of results. The correspondence between COVERS language and the Unified notation is outlined.

1. Introduction

The domain that we consider in this paper includes systems of various nature and scale that can be characterized as:

- Non-terminating  
- Continuously interacting with the environment  
- Having discrete, event-driven behavior  
- Consisting of concurrent interacting components  
- Having timeliness an important design issue.

Among others, this domain includes:

- Distributed algorithms and applications  
- Client/server systems  
- Communication systems  
- Computer systems and components  
- Networks  
- Real-time control systems

The designer of such a system can benefit from using its executable model throughout the whole design cycle in several ways. At the first place, the model will help to overcome the “sequentiality” of human's way of thinking and to understand what actually happens in the complex concurrent system. Specific correctness issues, such as

- Deadlocks  
- Starvation  
- Racing  
- Critical sections  
- Real-time temporal properties

can be tested as well as the general functionality. And of course, the designer can investigate the system performance, including

- Response times  
- Message latencies  
- System throughput  
- Resource utilization

In particular, for distributed software applications the executable model will allow to predict the effect of deployment topology on the application performance and find out how good does the application scale.

The today’s user has a choice of several commercially available tools supporting modeling and discrete-event simulation, BONeS Designer [1], SES Workbench [2], OPNET [3], Statemate [4] being the most known. Each tool offers some modeling framework: [block diagrams + resources + finite state machines] in BONeS, [transaction lifecycles + resources] in SES, [network topology + network node structure + finite state machines] in OPNET, [module charts + activity charts + statecharts] in Statemate.

While based on some useful ideas, these frameworks are overloaded with special non-standard notions, which makes it hard to integrate modeling and simulation into the system design process. None of the frameworks meet the modern requirements of complex system engineering, like object-orientation or reusability of components. Last, but not least, the mentioned tools are typical UNIX monsters: expensive, run on expensive machines, trying to be self-complete and, as a result, incompatible with anything.

The existing O0 design notations, such as, for example, the Unified notation [9], could serve as a good foundation for the proper modeling and simulation
Unfortunately, they do not have “enough semantics” to produce executable specifications.

The main goal of COVERS project at St.Petersburg Technical University is to bring modeling and simulation closer to the system design process by developing:

1. Open, flexible object-oriented modeling methodology based as much as possible on commonly accepted standards and having minimum special notions.
2. Graphical simulation tool with intuitive user interface running on highly available platform to support this methodology.

This paper considers the results we have by winter, 1997. The preliminary results regarding the model semantics can be found in [7].

2. The modeling language

The main construction unit of COVERS model is called active object. Active objects are independent concurrently-active event-driven logical machines. Depending of the level of abstraction, an active object can represent, for example:

- Piece of hardware
- UNIX process
- Human user
- WAN link
- Client or server
- Physical object

Active objects interact using asynchronous message passing. Messages are sent and received through ports. The set of public ports constitute the object interface.

Active objects may encapsulate other active objects to any desired depth. The encapsulated objects run concurrently with each other and with the container object. They can export their ports to the container object interface.

During its lifetime, the active object performs operations in response to the external or internal events and conditions. In case the event- and time-ordering of operations is important, the behavior of the object is described as a statechart.

An active object is a C++ object. Therefore, active objects can do anything C++ objects can do: have data members and member functions, inherit properties from other objects, call functions of each other, etc.

COVERS model is designed on the basis of active object classes. The specification of class includes its structure (ports, encapsulated objects, connections), behavior (states, transitions, etc.), data members and member functions.

2.1 Structure

The notation and the terminology used here was influenced by the ROOM approach [10].

Figure 1 shows the structure of a simple Server object. Server contains objects representing two hardware resources, CPU and RAM, and two software processes which access these resources. Whenever a new transaction arrives at LocalPort, the server activates Main object. The latter creates a new instance of Process object. Process executes the transaction (i.e. accesses CPU and RAM and, probably makes a remote request through a WANPort), replies to Main and then deletes itself.

Figure 1 Object structure

This example shows two important features of COVERS model:

- Dynamic creation and destruction of objects. Any object can be created and destroyed while the model is running.
- Replication of objects. Replicated object represents a collection of objects of the same class.

Figure 2 Modeling regular structures
Replication is especially useful when you are modeling a regular structure of objects, such as vector, chain, ring, mesh, torus, cube, etc., and the size of the structure is a parameter of the model. For example, the user can create the cell torus for the game “Life” of arbitrary size, see Figure 2.

2.2 Interaction

COVERS supports asynchronous message passing between objects. Asynchronous means that the sender can send a message at any moment, irrespective of the receiver (or receivers) state, and the control returns to the sender immediately after the message has been sent. Messages are sent and received through ports. An object sends a message to the port rather than to the other object. The set of destination objects is defined by the port connection topology.

There are end ports, through which the object communicates with the external and/or encapsulated objects, and relay ports, which export ports of the encapsulated objects (or private end ports) to the parent object interface. They normally just act as repeaters. Both end and relay ports can be divided into input ports, output ports, and input/output ports.

The mechanics of message passing is the following. When an active object wants to send a message, it sends it to the output end port. The latter immediately forwards the message to all ports connected to it (message multiplexing occurs if needed). When a message finally (maybe, having gone through several relay ports) gets to an input end port at the other active object, it is stored in the queue and object gets notified. The sender may then proceed without waiting for the receiver to consume the message.

The temporal semantics of the message passing is: once sent, the message gets to all destination ports in zero time (although, it may be consumed later). A single act of message passing is atomic, i.e. no other events can happen amid its execution. The communication latencies, when needed, are modeled at the level of communication objects (networks, channels, etc.).

The basic functionality can be changed by the user in any desired way by subclassing from COVERS port classes. This gives the flexibility, allowing to model, for example, shared data objects, message passing with addresses, filtering of messages, synchronous message passing, etc.

2.3 Behavior

During its lifetime, an active object performs operations in response to the external or internal events and conditions. The existence of state within an active object means that the order in which operations are invoked is important. For some objects, this event- and time-ordering of operations is so pervasive that we can best formally characterize the behavior of such objects in terms of a state transition diagram [8]. A state transition diagram is used to show the state space of a given active object class, the events that cause a transition from one state to another, and the actions that result from a state change.

As an object behavior notation COVERS supports statecharts based on David Harel’s Statecharts [6] - a simple yet highly expressive approach that is far superior to conventional flat state machines. Unlike Harel’s, COVERS statecharts are sequential. The active object whose behavior is specified as a statechart has a single location of control at any time (although it may run concurrently with the encapsulated objects). COVERS supports the following extensions to the conventional state machines: hyperstates, conditional states, history and deep history states.

Figure 3 shows the behavior of a hypothetical channel. It has two main modes: Normal and Crashed. In the Normal mode, while NotFull, the channel accepts incoming messages - transition MsgIn. For each message the channel schedules the exit event. If the channel becomes Full, it stops accepting until some other message exits the channel, transition MsgOut. Obviously, the exit event may occur irrespective of whether the channel is full or not - that is why MsgOut starts at the hyperstate Normal. At any time during the normal operation the channel may crash - see the transition Crash leading to the Crashed state. Assume that upon recovery the channel is able to restore the messages that were in the process of transmission. Then it should return to the last visited state in the Normal hyperstate - that is modeled by the transition Recovery leading to the history state inside Normal.

The visual appearance of statecharts and the semantics behind hyperstates, etc. is more or less standard and well
understood, whereas the attributes of states and transitions vary from implementation to implementation. In COVERs each state (primitive or hyperstate) has entry action and exit action, and each transition has guard, delay and action associated with it, see Figure 4.

Guard: Boolean expression
Delay: Expression of type TTime
Action: C++ code

Figure 4 Attributes of states and transitions

Guard is a Boolean expression which (semantically) is tested continuously while the object is in the transition’s source state. Presence of an input message, change of the shared object, or some complex condition on several port queues can be specified as a transition’s guard. In other words, in COVERs you write a predicate that expresses a stimulus (or event) the object wants to react to, and put this predicate into the guard of the transition.

A transition without a delay becomes enabled as soon as its guard becomes true. Transition with a delay becomes enabled as soon as its guard has been continuously true for exactly the delay period. Transition can not be enabled for a non-zero time interval: it should either be taken or be disabled.

Transition action and the appropriate state exit and entry actions are executed each time the transition is taken. Transition firing including the actions execution is atomic and takes zero time.

Not every active object has a significant event-ordered behavior. Some objects react the same way on the same set of events throughout their whole lifetime. The statechart can not be recommended as an adequate behavior specification, as it would trivially consist of one state and several transitions looping around it. For this case COVERs enables the user to specify a kind of the object “response table” of the form <event, handler function>.

2.4 COVERs language and the Unified Notation

Being an object-oriented simulation language, COVERs language can be also considered as a fully executable OO design notation. Let us outline the correspondence between COVERs and Booch method [8], that have now developed into the Unified method [9]. COVERs supports the following elements of the Unified notation (see Figure 5):

- Class diagrams expressing the physical containment relationship.
- Object diagrams where object links denote the asynchronous message passing.
- State transition diagrams (fully).

By no means are the properties and relationships of the classes and objects constituting COVERs model restricted to the listed above. These are only what we concentrate on when modeling and simulating the system, i.e. generate code, animate, visualize, etc.
encapsulation. The objects can, however, export their interfaces to the container object interface.

The other notable feature is the notion of port. An object sends a message to a port rather than to another object, and the (set of) destination objects is defined by the port connection topology - i.e. it is defined externally to the object.

With respect to concurrency semantics, COVERS active objects comply with the Booch's definition of active object: they encompass their own threads of control and are autonomous, meaning that they can exhibit some behavior without being operated upon by another object. In modeling practice, however, we meet objects serving just as containers for other objects, or objects that are unable to change state independently, being, therefore, passive according to Booch.

COVERS message passing is asynchronous. In Booch's terminology, the client sends the message to the supplier, the supplier queues the message, and the client then proceeds without waiting for the supplier. Other synchronization mechanisms are built on top of this one.

As long as COVERS model is fully executable, every element of it, be it state transition or port, has formally defined semantics with respect to time and atomicity - the feature lacking (or not required) in the OO design notations.

### 3. The semantics

The semantics of COVERS model is based on Timed Transition Systems [5]. Timed Transition Systems extend simple and economical interleaving approach to real time. There are two types of steps the model can make: time steps, when the time progresses, but the model state remains the same, and event steps when the model state changes instantly, see Figure 6. This allows us to handle correctly subtle aspects of concurrent system behavior, such as nondeterminism or racing.

![Figure 6 Semantical model](image)

In the situations when two or more events are scheduled exactly at the same time (like A, B, C and D) most of the simulation tools will always choose the first (in some deterministic order) alternative. COVERS will either make a random choice, or, in the interactive mode, leave it to the user. This ensures that a bigger part of the system state space is covered by a simulation, so it is more likely that an undesirable behavior is detected.

### 4. The tool

The complete picture of what COVERS does is shown in Figure 7.

![Figure 7 COVERS flowchart](image)
4.1 Code generation

COVERS model is completely mapped into C++. For each active object class specified by the user the model editor generates a C++ class. The encapsulated objects, ports, states and transitions become its data members. For example, the code generated from the specification of the server object Figure 1 looks like the following:

```c
class TServer: public TActiveObject {
  public:
    TServer() {
      Setup();
    }
    void TServer::Setup() {
      int i, count;
      LocalPort = new TInOut< TTransaction >();
      MainPort = new TInOut< TTransaction >();
      WANPort = new TInOutRelay();
      CPU = new TSharedServer();
      RAM = new TSharedResource();
      Main = new TMainProcess();
      for ( i = 0; i < N; i++ )
        Process.Add( new TProcess() );
      ConnectPorts( Main->CPUPort, CPU->Port );
      ConnectPorts( Main->RAMPort, RAM->Port );
      ConnectPorts( MainPort, Main->ServerPort );
      for ( i = 0; i < N; i++ )
        { MapPort ( WANPort, Process[i]->RightPort );
          ConnectPorts( Process[i]->MainPort, Main->ProcessPort );
          ConnectPorts( Process[i]->CPUPort, CPU->Port );
          ConnectPorts( Process[i]->RAMPort, RAM->Port );
        }
    }
}
```

The mapping into C++ makes the modeling technology very flexible, providing, for example, for the construction of template active object classes, i.e. those parameterized with the classes of other objects.

Before the generated C++ code is compiled, the user is allowed to modify it, so that arbitrary complex model structure or functionality can be achieved.

Arbitrary C++ modules can be included in COVERS project. This is particularly important in the so-called mixed-mode development process, when some part of the system is already implemented, whereas the other part needs simulation. For example, you have a distributed application and wish to try it on some non-existent computer system configuration. COVERS will take care of simulation of concurrency, communication latencies, resource sharing and so on, and the actual application will be linked to the model.

4.2 Model execution

During the interactive execution every bit of the graphical specification is animated and is accessible by the user, so that model debugging and analysis is done in terms of the original specification. COVERS highlights object interaction, current states and active transitions. Event queue, message queues are shown and controllable. There are also debugging tools programmable by the user:

- **Inspect.** Active objects and ports can display their current status information.
- **Log.** Active objects are able to write arbitrary textual information into their logs.
- **Runtime modification.** Data objects can be modified while the model is running. This is particularly convenient during the interactive experiments with the model, when the goal is not to collect the precise statistics, but to find quick answers to what-if questions.
- **Spy.** The user can define a function that will be called on each model step and test arbitrary conditions.
- **User-defined animated graphics.**

COVERS 3.0 C++ class library includes parameter classes, classes for statistics collection and reporting (for both discrete and continuous statistics), and data set classes.

Any model parameter can be specified by the user as iterated through a set of values. COVERS will perform as many simulation runs as required in order to cover the whole parameter space.

COVERS builds plots of three kinds: traditional plots, histograms and Gantt charts. In the latter ones the y-value is represented by the color of the stripe parallel to
x-axis. COVERS can also export data sets to other applications, such as MS Excel.

4.3 Libraries

COVERS enables the user to create libraries of reusable active objects. The libraries of frequently used objects are supplied together with COVERS. These includes:

- Resource models
  - CPUs
  - Memories
  - SCSI devices
- LAN models
  - Ethernet
  - 100Base VG
- Protocol models
  - Sliding Window
  - TCP
  - IP
- Various generic queues, channels, servers, etc.

For educational purposes we provide a number of classical examples of concurrent systems:

- Dining philosophers
- Mutual exclusion
- Distributed algorithms
  - Distributed termination
  - Clock synchronization
  - Leader election
- Concurrency control in distributed databases
  - Two-phase locking
  - Wound-wait
- Multicast protocols
  - ABCAST with Lamport’s linear time
  - CBCAST with vector time
and other models.

5. Conclusion

We have considered the modeling language based on active objects and the simulation environment COVERS. Let us summarize its features:

![Interactive model execution](image)
Object-oriented modeling language for building hierarchical active object models of arbitrary complexity and scale.

Statecharts is an advanced notation for object behavior description, accepted by all leading OO design methods, including the Unified Method.

C++. The model is completely based on and mapped into C++ - a standard, well-known language.

Formal semantics. COVERS simulation engine is build according to the formally defined language semantics and treats correctly the subtle aspects of concurrent system behavior, such as nondeterminism, atomicity or racing.

Open. COVERS is 100% open at the level of the C++ code it generates, allowing the user to modify it and to link arbitrary external modules. It can be used in the mixed-mode development process when the implemented part of the system coexists with the simulated one.

Libraries. The libraries of frequently needed objects are supplied with COVERS and can be easily extended. The user can create his own libraries of reusable components.

Cheap platform. COVERS runs on cheap, highly available platform: 486 or Pentium processor running MS Windows 3.1, Windows 95 or Windows NT.

User-friendly. COVERS enables the user to perform the whole modeling and simulation cycle within a single graphical environment ideologically close to Microsoft or Borland C++ development environments.

COVERS 3.0 with libraries and examples is available free over the Internet for non-commercial use. Our site is: http://dcn.nord.nw.ru.

Our current work is directed towards extending the active object libraries and providing the closer link with the OO design methods.

6. References


