JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 8, No. 1, January–February 2009

Activity Diagrams : A Formal Framework to Model Business Processes and Code Generation

A.K. Bhattacharjee Reactor Control Division, Bhabha Atomic Research Centre, Mumbai 400 085, email:anup@barc.gov.in

R.K. Shyamasundar School of Technology and Computer Science,Tata Institute of Fundamental Research, Mumbai 400 005,email:shyam@tcs.tifr.res.in

Activity Diagram is an important component of the set of diagrams used in UML. The OMG document on UML 2.0 proposes a Petri net based semantics for Activity Diagrams. While Petri net based approach is useful and interesting, it does not exploit the underlying inherent reactive behaviour of activity diagrams. In the first part of the paper, we shall capture activity diagrams in synchronous language framework to arrive at executional models which will be useful in model based design of software. This also enables validated code generation using code generation mechanism of synchronous language environments such as Esterel and its programming environments. Further, the framework leads to scalable verification methods.
The traditional semantics proposed in OMG standard need enrichment when the activities are prone to failure and need compensating actions. Such extensions are expected to have applications in modelling complex business processes. In the second part of the paper, we propose an enrichment of the UML Activity Diagrams that include compensable actions. We shall use some of the foundations on Compensable Transactions and Communicating Sequential Processes due to Tony Hoare. This enriched formalism allows UML Activity Diagrams to model business processes that can fail and require compensating actions.

1 INTRODUCTION

In model-driven development, models are used to describe user requirements, activities, information structures, components and component interactions of a system. These models govern the system development in a way that they can be transformed to ultimately to program code. UML is now the industry standard for describing software requirement specifications and design models [2]. It is a collection of graphical notations, each providing a particular view on the system being specified. Business processes based on workflows involve interaction and coordination between several services. The Unified Modeling Language (UML) and the Model Driven Architecture (MDA)provide a technology independent framework that can be used to model and specify composition of business processes.

Cite this document as follows: A. K. Bhattacharjee and R.K. Shyamasundar: "Activity Diagrams: A Formal Framework to Model Business Processes and Code Generation", in Journal of Object Technology, vol. 8, no. 1, January-February 2009, pp. 189-220 http://www.jot.fm/issues/issue_2009_01/article3/

One of the important modeling artifacts used in UML, is the Activity Diagrams (referred as UML AD)that are used to model sequence of actions as part of the process flow. It is used to model sequence of actions to capture the process flow actions and its results. It focuses on the work performed in the implementation of an operation (a method), and the activities in a use case instance or in an object. A simple activity diagram describing the order processing and account is shown in Fig. 1.

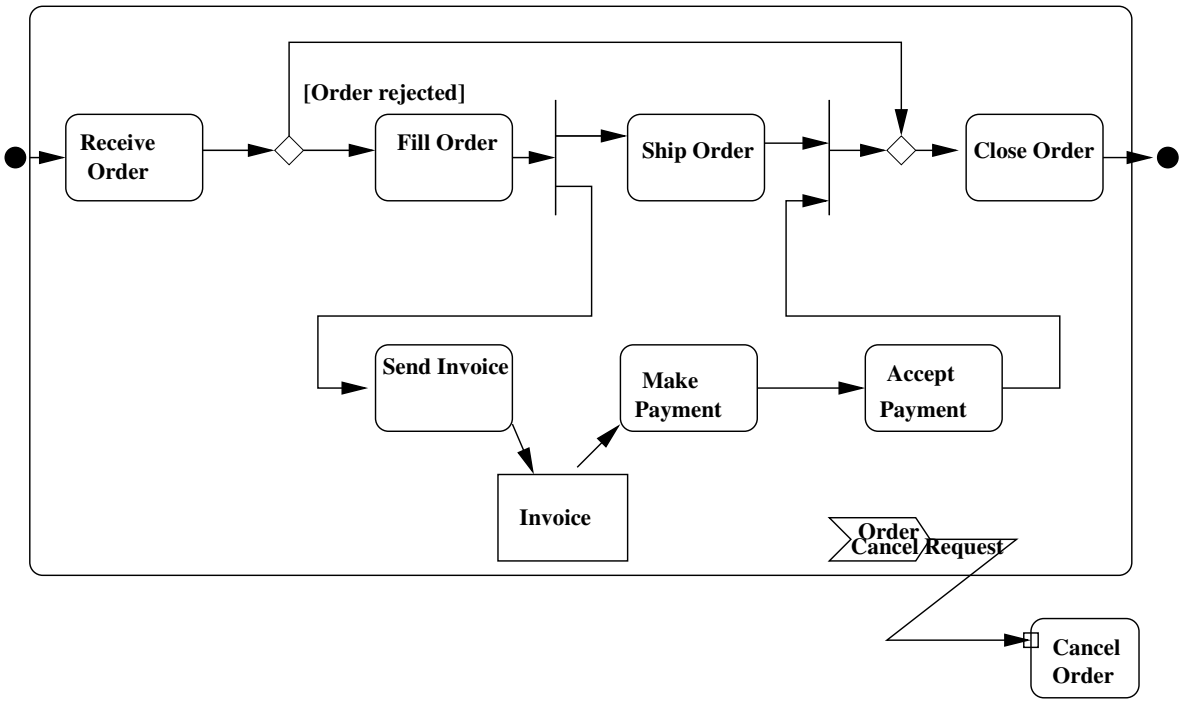
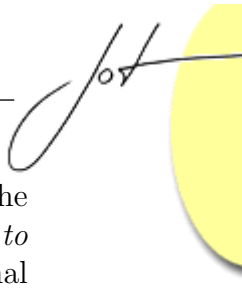


Figure 1: Simple Activity Diagram

Although the OMG document [1] provides an intuitive semantics of Activity Diagrams, it lacks a formal semantics required for analysis and automatic code generation. Hence, in the recent past there has been a lot of interest in giving a formal semantics to Activity Diagrams.

In the first part of the paper, we extend the process algebraic semantics of activity diagrams and propose a reactive formalism of Activity Diagrams of UML AD A shorter version of this work appeared in [27]. We use Esterel [13] language for description purpose. Our approach combines the requirement level and implementation level semantics. Further the notion of procedure call transitions as used in activity diagrams are captured nicely through the ‘‘run module’’ construct and one can specify the number of incarnations of the same module when called multiple times. Since it is based on Esterel, that has efficient code generation tools, the transformations can be used to realize a system directly from the model. Thus in our approach, we can not only reason about functional requirements of UML AD but also generate validated code automatically. This approach is useful for model based design of embedded systems.



In [7] it has been shown that UML AD can be used to model some of the workflows patterns identified in [6]. It is pertinent to ask *Can we use UML AD to specify business processes which are prone to failure?*. We opine that the traditional Activity Diagram needs to be enriched with additional constructs to enable us to model failures in any of the component processes. One of the advantage of having such a semantics for activity diagrams will allow modeling distributed workflows coupled with interruptible regions and evaluate their transitional state and behaviour for checking conformance to the requirement. Business Process Modeling Notation (BPMN) [3] has constructs to show failures and compensations, however we are not aware of a formal semantics of BPMN. On the other hand, the semantics of Activity Diagrams has been studied extensively in literature [24] and by enriching the constructs, it is expected to be useful. In the second part of the paper, we discuss the use of UML AD in modeling business processes.

The contribution of this paper is in the following

- Establishing a semantic mapping between UML AD to a synchronous language which allows validated code generation.
- Establishing the requirement of compensation actions in UML AD for modeling business processes.
- Enriching the constructs of UML AD with compensable actions [15] which enables modeling of failure in business processes.

The semantics of the additional constructs are based on CSP enrichment that can cater to failures in activities. Failures/exceptions are modeled as a part of the activities and is robust in the sense of CSP; as the failure action has also become an explicit action treated in a first-class manner and hence there is nothing like a real "exception". The proposed constructs can also be used to provide a theoretical framework for BPMN.

The paper is organized with an overview of the recent work in this direction in section 2. A description of various constructs of activity diagrams are given in section 3 and it's realization in a synchronous framework is given in 4. In section 5, a brief description of simulation and code generation based on the synchronous framework is presented. A brief insight into the requirement of activity diagram to model business logic is given in 6. Section 7 introduces the additional structures required to build compensating activities in modeling with a formal semantics. In section 8 a possible implementation of compensating activities is given in terms of Mode Automata. Verification approaches are presented in section 9 and the section 10 gives an outline of the future work.

2 RELATED WORK

To the best of our knowledge, the first formal semantics of UML AD was proposed by Eshuis [24]. A token flow semantics based on Petri Nets was proposed in [26]. Eshuis [24] proposes the semantics at the following two levels :*Requirement Level* and *Implementation Level*. The first level is based on Statechart like semantics and is transformed into a transition system for model checking by NuSMV. The second level is based on STATEMATE semantics of Statecharts extended with properties to handle data. The semantics covers activity charts of UML 1.5 but not of activity diagrams of UML 2.0[1]. Storrle [26] envisages a semantics by mapping activities into *procedural Petri nets*, which excludes data type annotations but includes control flow. He has defined mappings to *procedural Petri nets* to prevent multiple calls which otherwise would result in infinite nets. However these approaches do not address the automatic code generation as may be required in a tool driven environment. Semantics based on synchronous language was proposed in [27] which allows a validated code generation from the notation.

The suitability of activity diagrams for modeling business process has been argued in [7]. A process algebraic formulation of workflow is proposed in [8]. However these semantics cannot handle failures in activities and hence not suitable for modeling business processes. The extensions of Activity Diagrams proposed in this paper is inspired by Hoare [12] and is based on a flow composition language with a trace semantics introduced in [14]. A theoretical foundation of flow composition languages is given by Bruni in [25]. Fu et.al., presented an approach of converting BPEL Web services into guarded statement and further into PROMELA/SPIN for verification in [20]. Similar approaches based on Finite State Processes (FSP) [19] and CCS were presented in [21] and [23] . However they didn't consider failure and subsequent compensation. Verification approaches for incorporating compensating transactions were reported in [22, 28].

3 ACTIVITY DIAGRAMS: INTERPRETATION IN PROCESS ALGEBRA

An action is the fundamental unit of executable functionality in an activity. The execution of an action represents some transformation or processing in the modeled system, which could be a computer system or a process. An action may have sets of incoming and outgoing activity edges that specify control flow and data flow from and to other nodes. An action will not begin execution until all of its input conditions are satisfied. The completion of the execution of an action may enable the execution of a set of successor nodes and actions that take their inputs from the outputs of the action. The sequencing of actions are controlled by control edges and object

[1]It should be pointed out that UML 2.0 is a significantly re-engineered version of UML 1.5, particularly in the context of activity diagrams.

flow edges within activities, which carry control and object events respectively. An action can only begin execution when all incoming events are present. An action execution represents the run-time behavior of executing an action within a specific activity execution. When the execution of an action is complete, it offers events in its outgoing control edges, where they are accessible to other actions. Communicating Sequential Processes (CSP) [10] is a process algebra which is suited for modeling such process flow systems. One of the advantage of CSP is that one can make assertions about safety and correctness properties based on traces.

An interpretation of UML AD activity as CSP [10] processes is given below. The basic flow of activity is defined by the process PROC(P) as

$$\alpha PROC(P) = \{P.entry, P.in, P.exit\}$$
$$PROC(P) = P.entry \rightarrow P.in \rightarrow P.exit \rightarrow SKIP$$

The process PROC(P) first performs the event $P.entry$ at the start representing the start of the activity P, the event $P.in$ represents the activity P is being performed and $P.exit$ represents the completion of activity. In addition to these events, we consider two more events: $\surd$ to notify the upper level activities about the completion of the activities in the present scope, † to notify a general trigger to higher level activities. An activity diagram is constructed as a legal combination of activity, start, stop state elements and merge, decision, fork and join relationship elements.

Synchronisations in Various Control Flow Patterns

A number of control flow patterns and their ability to be modeled in UML AD has been identified in [7], which define elementary aspects of control flow. These are also used as elementary control-flow in Workflow Management.

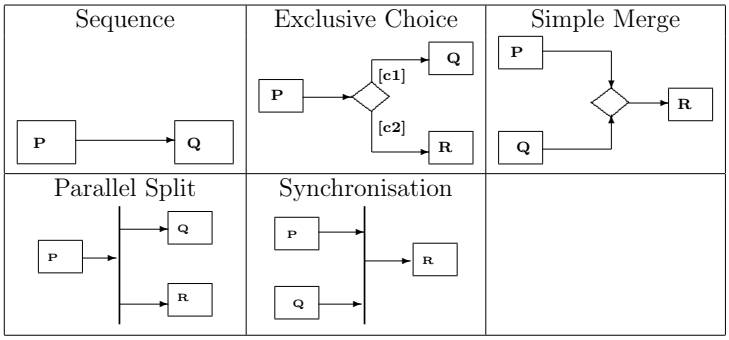


Figure 2: Basic Constructs of Activities

We provide the transition rules [10] of some of the basic control patterns shown in Fig.2 as processes in CSP. The terminal events $\Omega = \{\surd, \dagger, ©\}$ represent the different ways in which an activity may terminate. Successful termination is represented by the $\surd$ event, action denoted as $P \xrightarrow{\surd} \circ$. An interrupt event is represented by the †

event and yielding is represented by the ©event. The interrupt and yielding events are described later.

Sequence We consider three cases
Process P does not terminate

$$\frac{P \xrightarrow{\alpha} P'}{P;\ Q \xrightarrow{\alpha} P';\ Q} \alpha \in \Sigma$$

Process P terminates normally

$$\frac{P \xrightarrow{\surd} 0 \wedge Q \xrightarrow{\alpha} Q'}{P;\ Q \xrightarrow{\alpha} Q'} \alpha \in \Sigma \cup \Omega$$

Process P Terminates abnormally

$$\frac{P \xrightarrow{\omega} 0}{P;\ Q \xrightarrow{\omega} 0} \omega \in (\Omega - \{\surd\})$$

Exclusive Choice Condition $c_1 \subset \alpha$

$$\frac{P \xrightarrow{\surd} 0 \wedge Q \xrightarrow{\alpha} Q'}{P;\ Q \square R \xrightarrow{\alpha} Q'} \alpha \in \Sigma$$

Condition $c_2 \subset \alpha$

$$\frac{P \xrightarrow{\surd} 0 \wedge R \xrightarrow{\alpha} R'}{P;\ Q \square R \xrightarrow{\alpha} R'} \alpha \in \Sigma$$

Parallel Split

$$P;\ Q \parallel R$$

$$\frac{P \xrightarrow{\surd} 0 \wedge Q \xrightarrow{\alpha} Q' \wedge R \xrightarrow{\alpha} R}{P;\ Q \parallel R \xrightarrow{\alpha} Q' \parallel R} \alpha \in \alpha Q$$

$$\frac{P \xrightarrow{\surd} 0 \wedge Q \xrightarrow{\alpha} Q \wedge R \xrightarrow{\alpha} R'}{P;\ Q \parallel R \xrightarrow{\alpha} Q \parallel R'} \alpha \in \alpha R$$

The rules for *simple merge* and *synchronization* are expressed in terms of the above rules.

Simple Merge

$$P \square Q;\ R \equiv (P;\ R) \square (Q;\ R)$$

Synchronization

$$P \parallel Q;\ R$$

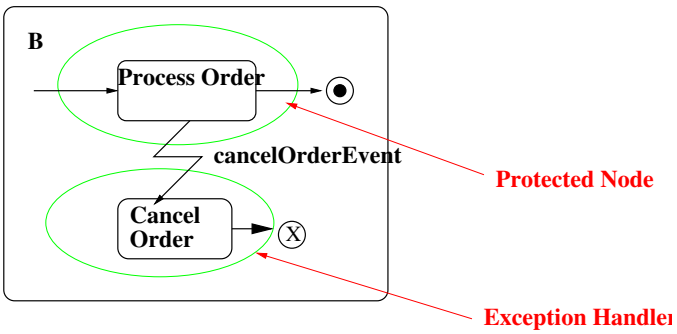


Figure 3: Activity with Exception Handlers

Handling Interrupts

Although not shown in [6], it is possible to break the flow in an activity diagram by an asynchronous interrupt event. If activities are sequential, the exception causes an immediate transfer of the flow of control. An interrupt handler may be used to catch interrupts: in $P \triangle Q$, an interrupt caused in P triggers execution of the handler Q. It follows that a trace of $(P \triangle Q)$ is just a trace of P up to an arbitrary point when the interrupt occurs, followed by any trace of Q.

$$\alpha(P \triangle Q) = \alpha P \cup \alpha Q$$

$$traces[\![P \triangle Q]\!] = \{s \frown \langle\dagger\rangle \frown t \mid s \in traces[\![P]\!] \cap \Sigma^* \wedge t \in traces[\![Q]\!]\}$$

The control flows to interrupt handler from the first process which is caused by the † event as shown below:

$$\frac{P \xrightarrow{\alpha} P'}{P \triangle Q \xrightarrow{\alpha} P' \triangle Q} \alpha \in \Sigma$$

$$\frac{P \xrightarrow{\dagger} 0 \wedge Q \xrightarrow{\alpha} Q'}{P \triangle Q \xrightarrow{\alpha} Q'} \alpha \in \Sigma \cup \Omega$$

$$\frac{P \xrightarrow{\omega} 0}{P \triangle Q \xrightarrow{\omega} 0} w \in (\Omega - \{\dagger\})$$

Fig. 3, shows how exceptions can be raised in activities and in CSP and it can be written as $ProcessOrder \triangle CancelOrder$. If interrupts could be nested i.e if $P \triangle Q$ could be interrupted and R is the handler then it could be specified as $(P \triangle Q) \triangle R)$. Generating an interrupt inside the activity diagram to interrupt other activities is shown by a special process called $THROW$ which generates the interrupt event † and terminates. On occurrence of the interrupt event control flow in the activity represented by process P will be transferred to Q. In case the interruption is caused by an external event (†) (not inside P) it is denoted as

$$\dagger \notin \alpha P P \triangle_{\dagger} Q = (P \triangle (\dagger \rightarrow Q))$$

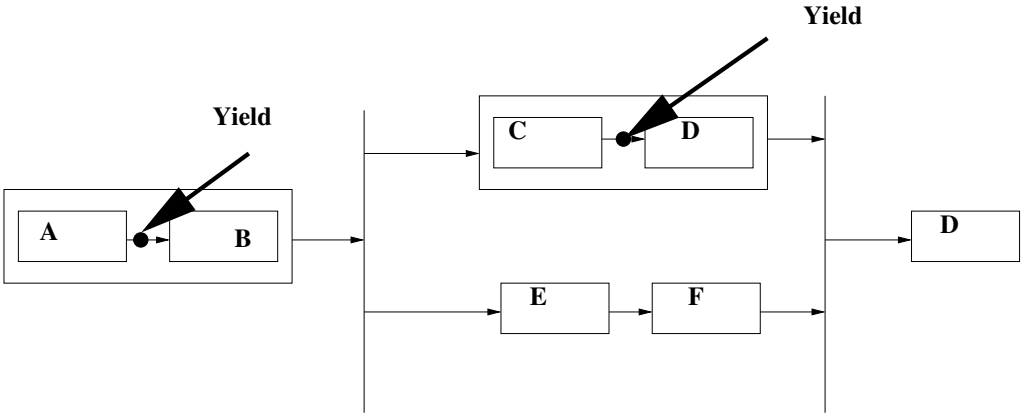


Figure 4: Activity with Yielding Points

An activity, which is ready to yield to an interrupt is to be indicated by local checkpoint where the state of the process is known and can be restarted. The behaviour of such activities is similar to check-pointed transaction. A local snapshot of the activity as shown in fig. 4 is taken through the primitive YIELD operator. These are the points where the local state of the activity is completely known. For example, P YIELD Q is willing to yield to an interrupt in between the execution of P and Q but not during P or Q. This is similar to the Hoare's *restart with checkpoint* operator [11]. The transition rules are specified as

$$\frac{P \xrightarrow{\alpha} P'}{P \text{ YIELD } Q \xrightarrow{\alpha} P' \text{ YIELD } Q} \alpha \in \Sigma$$

$$\frac{}{P \text{ YIELD } Q \xrightarrow{\dagger} Q \text{ YIELD } Q}$$

$$\frac{}{P \text{ YIELD } Q \xrightarrow{\copyright} P \text{ YIELD } Q}$$

The effects of terminal events on the special processes (**0** is the Null process) and their composition are defined below

$$\begin{array}{|l} SKIP \xrightarrow{\surd} \mathbf{0} \\ THROW \xrightarrow{\dagger} \mathbf{0} \\ THROW;\ P = THROW \\ THROW \triangle P = P \end{array}$$

4 SYNCHRONOUS FRAMEWORK FOR ACTIVITY DIAGRAMS

In this section, we provide an implementation [27] of the activity diagrams defined above in a synchronous framework. Synchronous framework is based on the perfect synchrony hypothesis: *the system reacts instantaneously to events producing outputs*

along with the input compiling away the control commands. Synchronous languages are based on this hypothesis and model reactive systems effectively and have a sound and complete semantics. One of the distinct advantages of using synchronous languages for specifying reactive systems is that the description of the system analyzed or validated is very close to implementation. One of the oldest languages in the family of synchronous languages Esterel has good developmental facilities such as efficient code generating compilers, verifiers etc. For these reasons, we have chosen Esterel as the underlying language for description of activity diagrams. A brief characteristics of Esterel is given in Appendix. The synchronous model for the Activity Diagrams is represented as a collection of transformation rules for each construct of the Activity Diagrams. In this paper, we are concerned with the Intermediate Level of Activity Diagrams that include control and data flow and decisions. A basic `ActivityNode` is modeled by an Esterel module named after the node. The invocation of the activity is modeled by instantiating the module using the `run module` construct.

A basic ActivityNode can invoke an asynchronous task which can handle system specific functions and can be modeled by an Esterel task statement such as `exec taskA ()() return ExitA`, where taskA is the external process performing the actual action written in the host language. The completion of the task is signaled by emitting the signal `ExitA` referred as a return signal. A return signal cannot be internally emitted by the program. In our model we ignore the external action for the purpose of simplicity.

Each activity node has the following set of signals associated with it.

- `EntryS` is the signal emitted when a particular activity node is entered.
- `InS` is the signal emitted when an action in a particular activity node is being performed.
- `ExitS` is the signal emitted when a particular activity node is completed.

We also assume that there is a root activity node which contains and controls the sequencing of the activity nodes through the activity edges. In the example shown in Fig. 5 the module `simpleActivity` performs the task of passing control tokens from the activity `sendPayment` to the activity `receivePayment`. The activity node `simpleActivity` is the root activity controlling the activities `sendPayment` and `receivePayment`. The activities `sendPayment`, `receivePayment` and `simpleActivity` in the above example, can be interpreted through the Esterel fragments shown in the Fig.5.

Merge Node:

A merge node (cf. Fig. 6) is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among

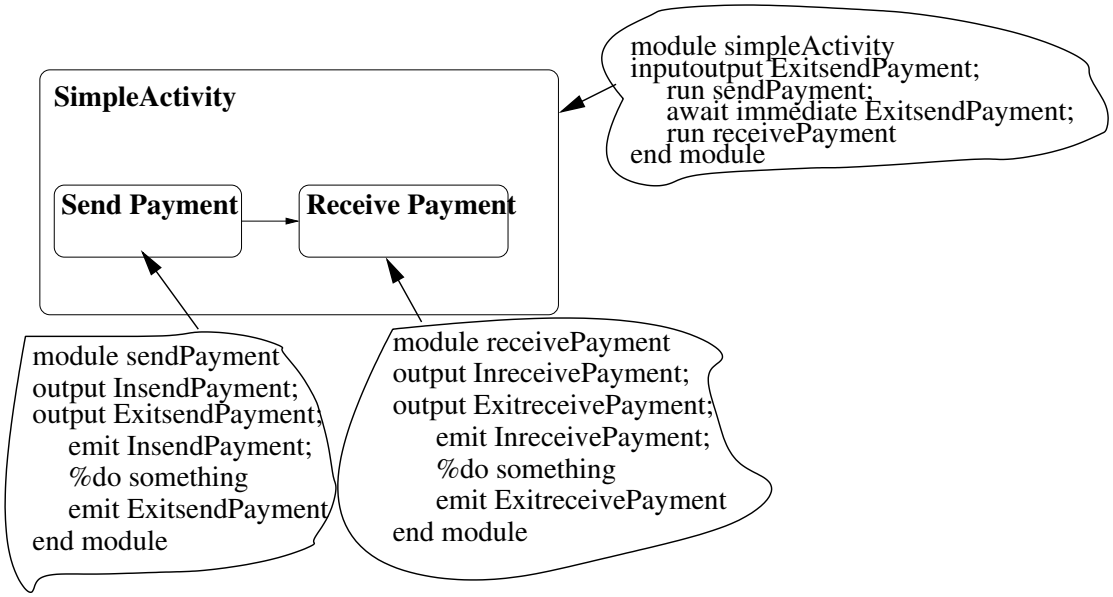


Figure 5: Simple node

alternate flows. It has multiple incoming edges and a single outgoing edge. It can be described as follows

module *mergeNode*
run *A*% the module A implements activity A
||
run *B*% the module B implements activity B
||
await *ExitA* ;
run *C*% The module C implements activity C
||
await *ExitB*
run *C*% The module C implements activity C
end module

Here the activities A and B are started concurrently, but whichever activity completes earlier, starts the activity C. If activity A and B completes together, then two instances of C would be running at the same time. This interpretation is in line with recent OMG document [1].

Decision Node:

A decision node (cf. Fig. 7) is a control node that chooses between the outgoing flows. It has one incoming edge and multiple outgoing edges. It can be described

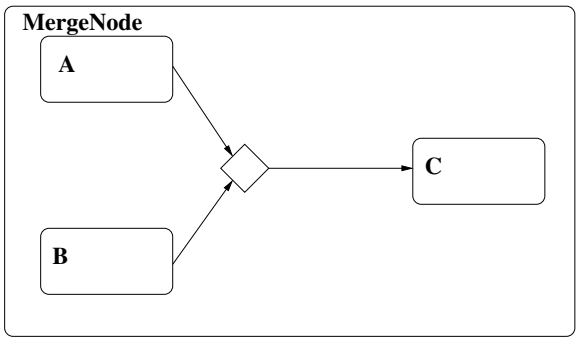


Figure 6: Merge Node

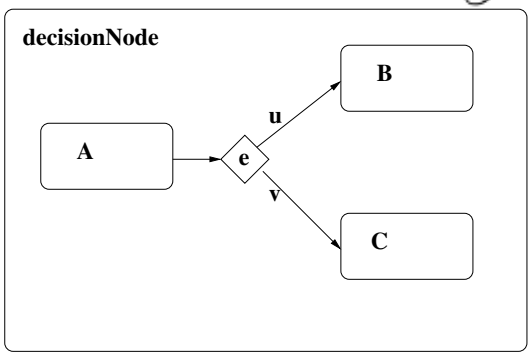


Figure 7: Decision Node

by the following Esterel fragment.

```
module decisionNode
    var e in
   run A ;
   if e = u
      run B ; % e is the guard which if has value u then run B
    else if e = v
            run C ; % e is the guard which if has value v then run C
     end
   end
end module
```

Here after the activity A completes, the control passes to activity B or C depending on the guard condition e being equal to u or v respectively.

ForkJoin Node:

A forkJoin node (cf. Fig. 8) is a control node that splits a flow into multiple concurrent flows. It has one incoming edge and multiple outgoing edges. Tokens arriving at a fork node are duplicated across the outgoing edges. Tokens offered by the incoming edge are all offered to the outgoing edges.

The forking and joining of activities can be described by the following Esterel fragment.

```
module forkJoinNode
   run A% run activity A
      [
```

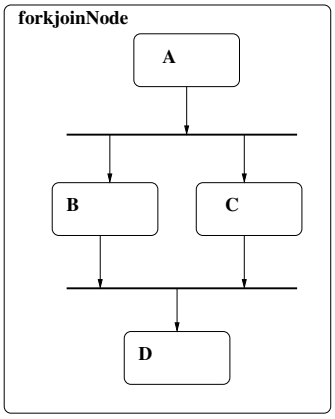


Figure 8: Fork Join Node

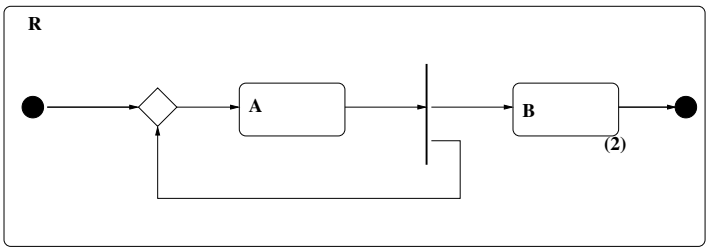


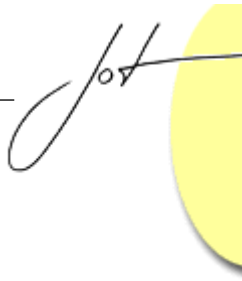
Figure 9: Reentrant Node

run *B*% run activity B
||
run *C*% run activity C
]
run *D*% run activity D
end module

Here after the activity A completes the activities B and C are started concurrently. Once both of B and C are complete, D is started. If concurrent activities are not modeled carefully this may lead to problem. Let us consider the case as shown in the Fig. 9. Here completion of A forks A once again with B. Thus, a possible run of the system is $A \rightarrow AB \rightarrow ABB \rightarrow \cdots$. That is there can be an infinite incarnation of B. This causes problem with verification because of unboundedness of states.

If we need to consider finite number of instances, we can use the parallel construct in Esterel to specify a finite number of concurrent activities. This is an advantage of the model, where one can specify the number of instances of the same activity which could be forked simultaneously. This closely maps to Workflow Management Systems, where one would specify the maximum number of such concurrent instances of an activity. The Esterel model of the activity diagram shown in Fig. 9 is shown below. The module R is the coordinating module for A and B. In this model we assume that there could be at most two instances of activity B as shown by the two modules named B1 and B2 in the code. In Fig.9 the number shown in bracket indicates the maximum possible number of instances of activity B. Here we assume calling external tasks as final activities for ActivityNodes A and B.

module *A* :
output *InA* ;
return *ExitA* ;
task *activityA ( ) ( )* ; % external asynchronous task declaration



```
    exec activityA ( ) ( )  return ExitA% external action
      ||
    abort
        sustain InA ; % indicates module A is active
    when  ExitA
end module
module B :
    return ExitB ;
    output InB ;
    task activityB ( ) ( ) ; % external asynchronous task declaration
    exec activityB ( ) ( )  return ExitB% external action
      ||
    abort
        sustain InB ;
    when  ExitB
end module
module R :
    return ExitA,ExitB1,ExitB2 ;
    input InA, InB1,InB2 ;
    task activityA ( ) ( ) ; % external asynchronous task
    task activityB ( ) ( ) ; % external asynchronous task
    input start ;
    signal b1b2, free  in
        loop
            await [  start  or ExitA  ];
            present  free  then [
                                    abort
                                        run  A
                                    when  ExitA
                ]
            end
        end
          ||
        loop
            present [  not InB1  ] then % First instance of B
                [
                    await  ExitA ;
                    run  B1/B  [ signal  ExitB1/ExitB,InB1/InB  ]
                ]
              else [  present  not InB2  then
                        [    % Second instance of B
                            await  ExitA ;
                            emit b1b2 ;
                            run  B2/B  [ signal  ExitB2/ExitB,InB2/InB  ]
```

```
                    ]
                else [
                        await [ ExitB1 or ExitB2 ];
                        emit start
                    ]
                end
            ]
        end present
    end
     ||
    loop
        await start ;
        abort
            sustain free% free is on when B1 is active but B2 is dormant
        when b1b2
    end
  end
end module
```

Since each run B produces a separate instance of the task associated with the activity B, several simultaneous instances of activity associated with B can exist. In this case one should specify the number of instances of such activities. The model here shows capability of running two identical activities concurrently.

Modeling Exception:

Fig. 10, shows the exception in an activity diagram. The node which is aborted due to the exception is called the protected node and the receiving node is the exception handler node. An exception handler is an element that specifies a body to execute in case the specified exception occurs during the execution of the protected node. In Fig. 10, Activity Node `ProcessOrder` is the protected node and `CancelOrder` is the exception handler and `CancelOrderEvent` is the exception input. This can be modeled in Esterel as shown below..

```
module B
  input cancelOrderEvent, ExitProcessOrder ;
  trap T in
      run ProcessOrder
       ||
      abort
          loop
              await cancelOrderEvent ; % Watch exception event
```

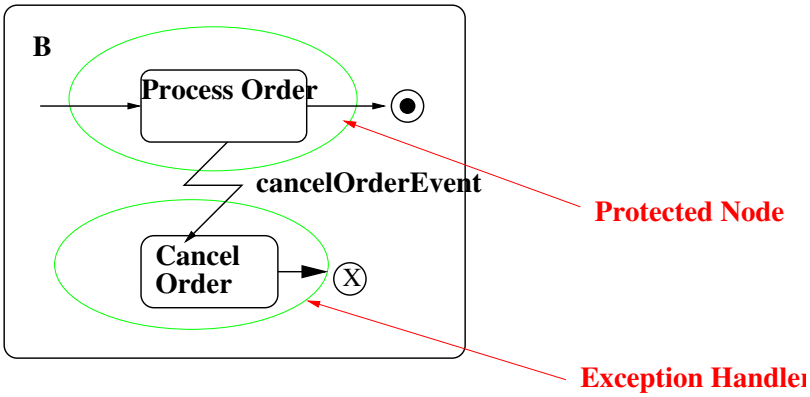


Figure 10: Exception Node

```
            exit T
        end
    when ExitProcessOrder
handle T do
    run cancelOrder% Exception Handler
end
end
```

Here the activity `ProcessOrder` is preempted and the the activity `cancelOrder` is executed on raising the exception event `cancelOrderEvent`.

Activity with Data and Nesting

In many instances one *ActivityNode* may need to pass a data to another ActivityNode for processing by the *Activity* performed at that ActivityNode. For example if P and Q are two ActivityNodes and P is required send a data X to Q as shown in Fig.11 then this can be modeled using the mechanism shown below. The ExitS signal emitted by the activity node S is used for synchronizing the fact that the data token is available at the end of activity P.

```
module main
inputoutput X:type % X is the data which is passed between activities
    run P(X)
    await immediate exitP
    run Q(X)
end module
module P
output X:type
...
```

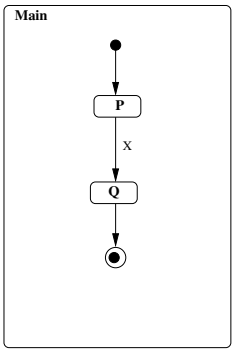


Figure 11: Object node with data

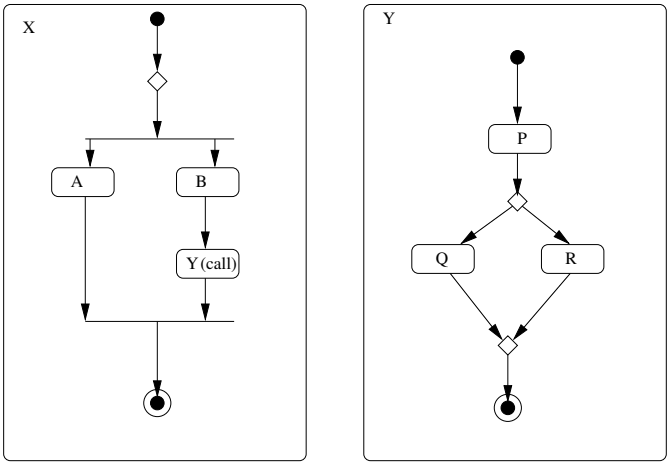


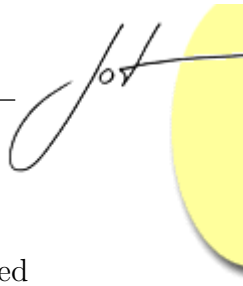
Figure 12: Activity with Nesting

```
module X                    module Y
...                                 ....
          run A                     run P;
                                    if e = u then
          run B;                                    run Q
          run Y                     else if e = v then
          ...                                       run R
          ...                       end
end                         end
```

```
    emit ExitP
end module

module Q
input X:type
task QActivity()(); % declaration of asynchronous task
...
     exec task QActivity(X) return ExitQActivity;
...
end module
```

In our model, Activity Diagrams with nested *call* can be modeled naturally. Let us assume that one activity Y is nested in another activity X as a `call Y` action in the activityNode C of X shown in Fig. 12. This can be modeled by using the *run* Y construct of Esterel. The following Esterel fragment describes the nested call of the Fig.12.



Communication in Activity Diagrams

The notion of communication between two Activity Diagrams can be nicely modeled in the Communicating Reactive Processes (CRP) [18] framework. The CRP model consists of network $M1M2..Mn$ of Esterel modules, each having its own inputs and outputs and its own notion of instants. The network is asynchronous and the nodes communicate though synchronous channels. In this model, each M_i is an Activity Diagram each of which evolve locally with its own input and output and mutually independent notions of time [18]. Signals may be sent or received in activity diagrams through channels and is denoted by the common send and receive nodes. As an implementation model, one can think of an asynchronous layer (task) that handles rendezvous by providing the link between the asynchronous network events and node reactive events. The shared task can be called as channel. Fig. 13, shows a simple example of an activity diagram showing two component activities *PrintServer* and *PrintClient* communicating data (as files) through a channel. The CRP code for the same is shown below.

```
module PrintServer
input channel printq from PrintClient : FILE % CRP channel
......
    receive(printq,file) % send data file to printq
.....
end module
module PrintClient
output channel printq from PrintServer :FILE % CRP channel
...
    send(printq,file)    % receive data file from printq
....
end module
```

The send and receive [17] are communication primitives realizing the communication rendezvous between two locally synchronous programs. The primitive `send` blocks until sending data on the named channel succeeds and the primitive `receive` blocks until a communication succeeds on the named channel and the value assigned to the variable.

5 SIMULATION AND CODE GENERATION

Above we have shown how activity diagrams can be transformed into Esterel model. We are augmenting our previous work [16] to translate them automatically. The Esterel model can be simulated by using the *xes* interface. *Xes* is the simulator freely available along with the Esterel distribution. The simulator can be generated

by compiling the Esterel program with the xes library. The simulation gives the user a clear picture of the execution of the activity diagrams and checking conformance to requirement is easy. We are also building simulators directly in the domain of input activity diagrams whereby one can see the simulation graphically.

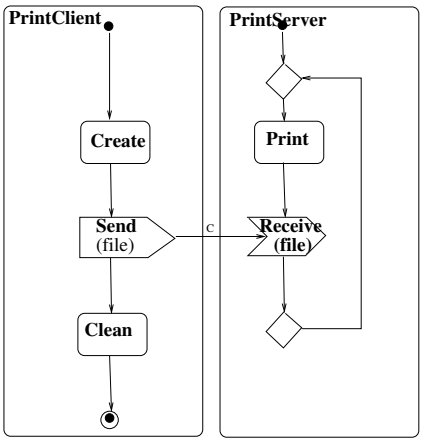


Figure 13: Object node with communication

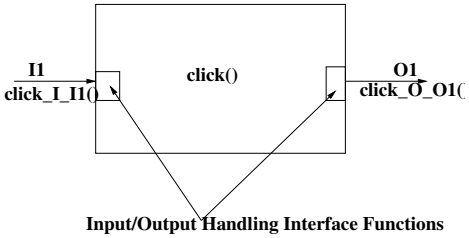


Figure 14: Activity to Code Mapping

Code Generation

There are two orthogonal levels of semantics, both indispensable: the intuitive level, where semantics must be natural and easy to understand, and the formal level, where the semantics is rigorously defined and fully non-ambiguous. Having formal semantics for the languages also makes code generators much easier to develop and verify. The translation process from Activity Diagrams to High Level Language (HLL) code like C is based upon sound proven algorithms that the Esterel code generators directly implement. By providing a formal semantics based on the synchronous paradigm and Esterel, it is easy to build correct code by construction, using Esterel-C/Java code generators. We assume Esterel-C code generator for further discussion.

For actual execution of the code , the generated code must also be linked with some extra layer of code that realizes the interface with the outside world which detects input events, read data and realizes output events and send data.If for example the module `click` should react to an input event, composed for example of one input tokens I1 as shown in Fig. 14. The sequence will include call to one automatically generated input C function `click_I_I1()` . This should be followed by call to the reaction function by executing the C code `click()`, followed by a call to output C function `click_O_O1()`.

The automatic code building process is achieved using the rules described above

1. Model the flow as an activity diagram model

2. Transform the model into the Esterel model following the rules as described above. These can be automated by encoding them in a model transforming algorithm similar to [16].

3. Describe interfaces as required by the Esterel modules regarding inputs and outputs.

4. The activities to be performed in the software `exec tasks` are to be encoded in the host language and operating systems.

6 MODELING BUSINESS PROCESSES AS ACTIVITY DIAGRAM

Let us now consider the activity diagram shown in Fig. 15, where activity B can fail. This activity B can be a service provided by a server. Since the UML AD can only model forward flow, it is not possible to show the actions required if the service provided by this activity fails.

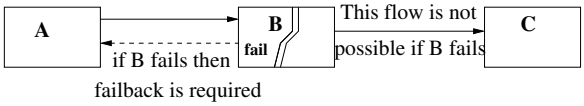


Figure 15: Activity with Failure

If the activity A is not successful because of some internal exception or an external condition, we must be able to undo the partial effect of actions executed in A. Let us now consider the activities required to process an order for which the activity diagram is shown in fig 16.

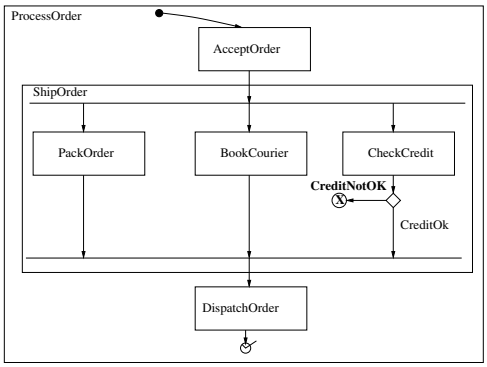


Figure 16: Order Processing

This is the classical book store problem where customers can order books over the web to the vendor. The vendor may not store all books and need some time in processing with other suppliers. However he can check the credit status of the customer with the bank. In case the activity `CheckCredit` reports a credit failure like `CreditNotOK` which should trigger the cancellation of the order, cannot be shown

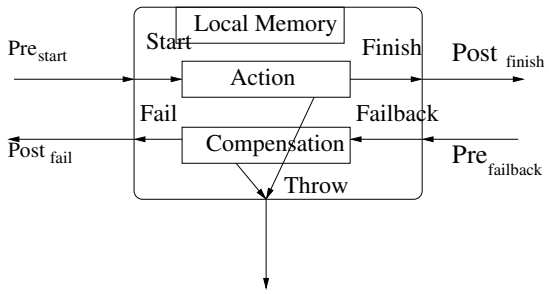


Figure 17: Activity with Compensation

using the traditional token flow semantics of UML Activity diagrams. This is the underlying motivation of this paper.

A business process as described above typically consists of steps (each of which may be refined in substeps) and each step is called an activity. The requirements of business processes modeling are to be able to describe the process map showing the flow in the activities, description of these activities, handling exceptions and failure.

7 MODELING FAILURES IN ACTIVITY DIAGRAMS

We propose to extend the syntax and semantics of UML activity diagram inspired by [12] and [15]. Here an activity is drawn as a box with two entry points and three exit points. The entries and exits are as shown in Fig.17. The box indicates an activity which may be composed of sub-activities but the interior components and connections of the box can be ignored from the outside. The entry and exit points of a compensable activity are activated in a standard sequential ordering. The normal entry point for an activity is at the start and failure leads to an exit along the exit labeled fail, which returns control to the compensation of the previous transaction. Successful execution ends with a finish, which will start the next activity in the sequence. If a subsequent activity fails triggering a failback, so that in this activity is able to compensate. If an activity detects that it can neither compensate nor succeed, it will allow the control to pass on the throw exit, which needs to be handled at the higher level as shown in Fig.18. After compensation, the activity exits by the failure arrow as before. In this sense, the compensable activities has a three way token flow.

For example, consider a simple activity whose input is X and which computes an output data Y such that $Y = X + X$. The compensating activity must be able to compute X such that $X = Y/2$. The action of compensation is to save a local snapshot of local state (values of variables) before change and restore it when required to compensate. This is the technique used in traditional transaction processing systems. The post condition of an activity at the finish edge must entail the precondition at the failback edge. $Post_{finish} \Rightarrow Pre_{failback}$. Similarly $Pre_{start} \Rightarrow Post_{fail}$.

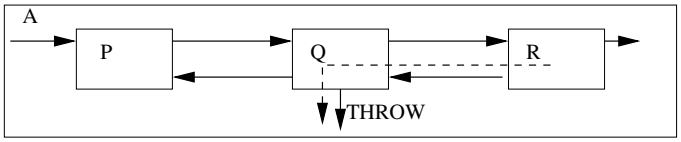


Figure 18: Composition of Compensating Activities

This is similar to what is supported at procedural level in BPEL4WS where the compensation handler can be invoked by using the `compensate` activity.

```
<compensationHandler>
activity
</compensationHandler>

<compensate scope=''ncname'' ? attributes>
standard block
</compensate>
```

The advantage of graphical notation like Activity Diagram is that it would be easier to capture the choreography in a graphical formalism than in a imperative language like BPEL.

Semantics of Activity Diagrams with Failures in Enriched CSP Framework

In order to support failed activities, we use compensation operators [15] and the *Activities* are classified into *standard* and *compensable activities*. A compensable activity has associated compensation actions which are invoked in case of a failure in the forward activities. A compensable activity consists of a forward behaviour and a compensation behaviour. In the case of an exception, activities will be executed to compensate the forward behaviour. The basic way of constructing a compensable activity is through the compensation primitive $P \div \bar{P}$, where P is the forward activity and $\bar{P}$ is its associated compensation. $\bar{P}$ should be designed to compensate for the effect of P and may be run after P has completed. The parallel and sequential composition operators for compensable processes are designed in such a way that ensures that after the failure of an forward activity the necessary activities are performed in an appropriate order to compensate the effect of already performed actions. Sequential composition of compensable processes is defined so that the compensations for all performed actions will be in the reverse order to their original sequence.

The compensation enabled activity $PP = P \div \bar{P}$ is composed of two standard processes. The first one is called forward process which is executed during normal execution and the second one is called the compensation of the forward process

which is stored for future use when it is required for compensation:

$$\frac{P \xrightarrow{\alpha} P'}{P \div \bar{P} \xrightarrow{\alpha} P' \div \bar{P}} \alpha \in \Sigma$$

If the forward activity terminates normally then the complete activity terminates with $\bar{P}$ a the result compensation. This is to say that the at the end of successful termination of present activity, the compensating activity $\bar{P}$ is installed.

$$\frac{P \xrightarrow{\surd} 0}{P \div \bar{P} \xrightarrow{\surd} \bar{P}}$$

If any forward activity terminates abnormally, then so does the complete activity, resulting in an empty compensation activity

$$\frac{P \xrightarrow{\omega} 0}{P \div \bar{P} \xrightarrow{\omega} SKIP} w \in \{\dagger, ©\}$$

A standard activity can be transformed into a compensable activity by adding to it an activity, which actually does nothing (SKIP). We use P,Q to identify standard activities and PP,QQ to identify compensable activities.

PP	$::= P \div \bar{P}$(compensation pair)
	$\mid SKIPP = SKIP \div SKIP$
	$\mid THROWW = THROW \div SKIP$

Standard activities can be constructed with the CSP operators for choice, sequencing and parallel composition. The compensation enabled activity $PP = P \div \bar{P}$ is composed of two standard processes. The first one is called forward process which is executed during normal execution and the second one is called the compensation of the forward process which is stored for future use when it is required for compensation: $\frac{P \xrightarrow{\alpha} P'}{P \div \bar{P} \xrightarrow{\alpha} P' \div \bar{P}} \alpha \in \Sigma$ If the forward activity terminates normally then the complete activity terminates with $\bar{P}$ a the result compensation. We say that the at the end of successful termination of present activity, the compensating activity $\bar{P}$ is installed.

$$\frac{P \xrightarrow{\surd} 0}{P \div \bar{P} \xrightarrow{\surd} \bar{P}}$$

If any forward activity terminates abnormally, then so does the complete activity, resulting in an empty compensation activity

$$\frac{P \xrightarrow{\omega} 0}{P \div \bar{P} \xrightarrow{\omega} SKIP} w \in \{\dagger, ©\}$$

If the activity $PP = P \div \bar{P}$ cannot progress either way due to an internal condition, it generates a throw which should be caught by an exception handler. This handles the three way flow

$$PP \triangle I_P = [P \div \bar{P}] \xrightarrow{\dagger} I_P$$

$$traces[\![PP \triangle I_P]\!] = \{s \frown t \mid s \in traces[\![P \div \bar{P}]\!] \wedge t \in traces[\![I_P]\!]\}$$

Let us consider $PP = P \div \bar{P}$ and $QQ = Q \div \bar{Q}$ as two compensable activities then the following rules define the sequential composition of compensating activities

$$\frac{PP \xrightarrow{\alpha} PP'}{PP;\ QQ \xrightarrow{\alpha} PP';\ QQ} \alpha \in \Sigma$$

if PP fails the whole activity terminates and the compensation activity of PP that is run.

$$\frac{PP \xrightarrow{\alpha} \bar{P}}{PP;\ QQ \xrightarrow{\alpha} P} \alpha \in (\Omega - \{\surd\})$$

However if QQ terminates normally after PP, the compensation of PP i.e $\bar{P}$ should be composed with the compensations from QQ i.e $\bar{Q}$. The reversal of process order is shown by $\langle \bar{Q}, \bar{P} \rangle$. This is shown by

$$\frac{PP \xrightarrow{\surd} \bar{P} \wedge QQ \xrightarrow{\surd} \bar{Q}}{PP;\ QQ \xrightarrow{\omega} \bar{Q};\ \bar{P}} (\omega \in \Omega)$$

A compensable activity PP can be converted into standard activity by defining a block $[PP] = P \div \bar{P} \backslash \alpha P \cup \alpha \bar{P} \cup \dagger$. Successfully completed PP represents successful completion of the whole transaction block and compensations are no longer needed. When the forward behaviour of PP throws an interrupt, the compensations are executed in the appropriate order and the interrupt is not observable outside the block. Parallel composition of compensable activities is defined in such a way that compensations for performed actions will be accumulated in parallel. We assume that each of the activities P and Q are not raising interrupt and not yielding to interrupt.

$$[P \div \bar{P} \parallel Q \div \bar{Q};\ THROWW] = (P \parallel Q);\ (\bar{P} \parallel \bar{Q})$$

$$[P \div \bar{P} \parallel Q \div \bar{Q} \parallel THROWW] = SKIP \Box (P;\ \bar{P})$$

$$(Q;\ \bar{Q}) \Box (P \parallel Q);\ (\bar{P} \parallel \bar{Q})$$

A typical behaviour concerning the stack of compensation activities is shown in Fig. 19. One of the safety requirement of such compensating activity diagram is that the stack of compensating activities must be empty at the end. Now let us consider the above activity diagram in Fig.16 to process orders which require compensation because of exceptions raised by the `CheckCredit` activity when sufficient credit does not exist. The modified activity diagram is shown in Fig 20.

The activities can be specified formally as

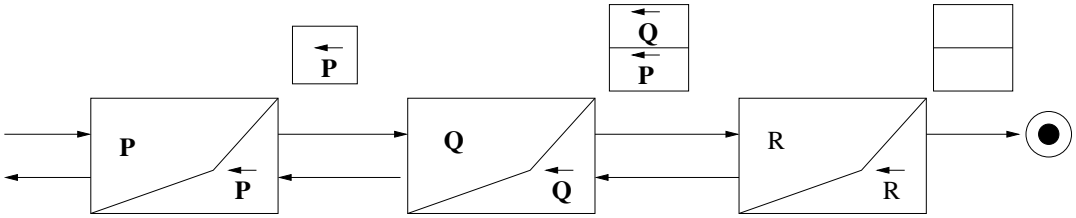


Figure 19: Stack of Compensation Activities

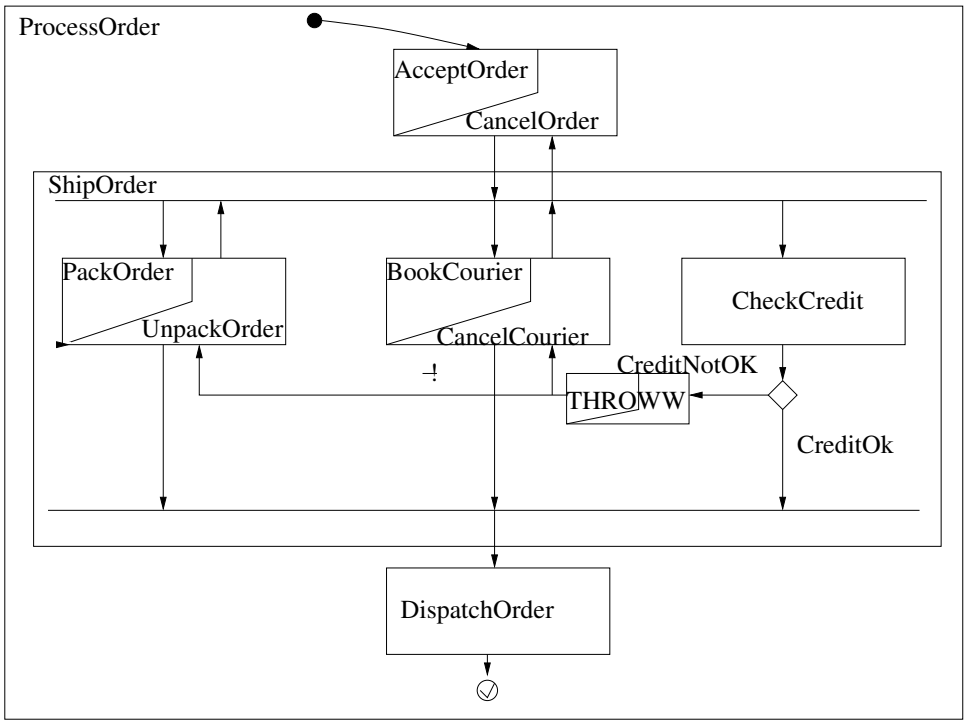


Figure 20: Activity with Compensation

$$\begin{aligned}\text{ProcessOrder} = (\text{AcceptOrder} \div \text{CancelOrder})\\ ; \text{ShipOrder}; \text{DispatchOrder}\\ \text{ShipOrder} = (\text{PackOrder} \div \text{UnpackOrder}) \parallel\\ (\text{BookCourier} \div \text{CancelCourier})\\ \parallel \text{CheckCredit} ; (\text{CreditOK} ; \text{SKIPP} \Box\\ \text{CreditNotOK} ;\text{THROWW})\end{aligned}$$

This shows the underlying formal description of the activity diagram with compensating constructs. The advantage is in that this can be subjected to analysis for showing certain desired properties of business logic. The model can be used also to construct an implementation from the description like that of [29].

8 IMPLEMENTATION MODEL FOR COMPENSATING ACTIVITIES

The compensating activity diagrams can be represented as a model in a synchronous framework based upon Mode Automata [32]. Mode Automata is a synchronous language which combines synchronous data flows with running modes. The compensating activity could be considered as having two modes: normal and compensating modes. The normal mode defines the activity in the forward direction and the compensating mode defines the activity which is run in case of a failure in the subsequent activities. Fig. 21 shows two compensating activities PP and QQ. The modes of PP are also shown as a Mode Automaton in bottom of the the Fig.21. The compensating activities PP and QQ are shown as two concurrent state machines. In the forward mode of P the variable x is incremented by 1. In case the forward activity of $QQ = Q \div \bar{Q}$ fails, it is compensated by the compensating mode $\bar{P}$ of PP. The actual action in each activity is written as a dataflow equation in the box. These could be the *tasks* as shown in earlier in the ESTEREL code. In Fig. 22 we show the composite Mode Automaton.

9 VERIFICATION

We only discuss the verification approach in case of conventional UML AD (i.e. without compensation). The model captures the operational semantics of activity diagrams. However it is not amenable to formal verification using model checking due to presence of asynchronous tasks invoked by the `exec` statements. For the purpose of verification, it is required to do a control abstraction of the Esterel models whereby we only retain the labels where the task is to be created. The derived model is thus converted into a pure Esterel program and one can perform a constructive causality analysis using the Esterel compiler option of *causal*. This model can then

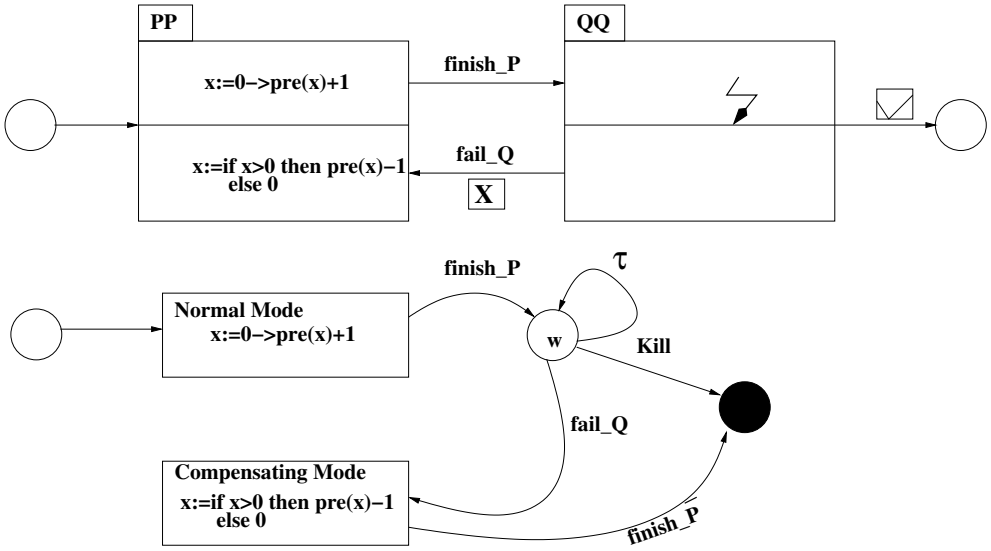


Figure 21: Mode Automata for Activity with Compensation

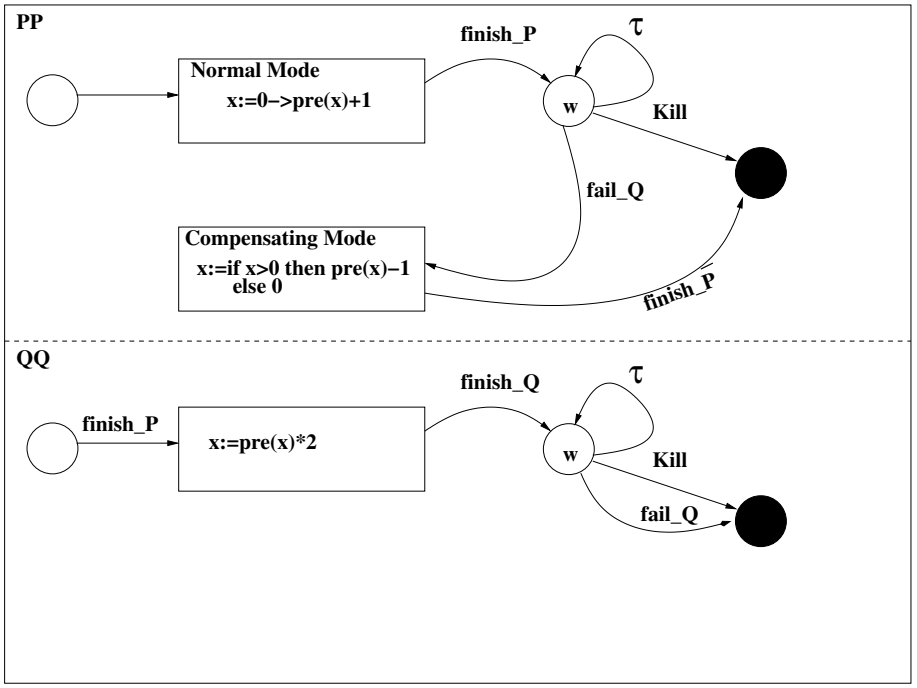


Figure 22: Composition of Mode Automata

be converted into an automaton in BLIF (Berkley Logical Interchange Format) format, which is accepted by the Esterel model checker *xeve*.

As an example, let us consider the activity diagram given in Fig. 9 with the following very simple safety property: *when both B1 and B2 activities are going on activity A cannot be started*. It is to be noted here that B1 and B2 are two incarnations of the activity B. This is assuming that there is no queuing of input. This could be verified by *xeve*. The screen shots taken from *xeve* are included here in Figs.23,24 for reference.

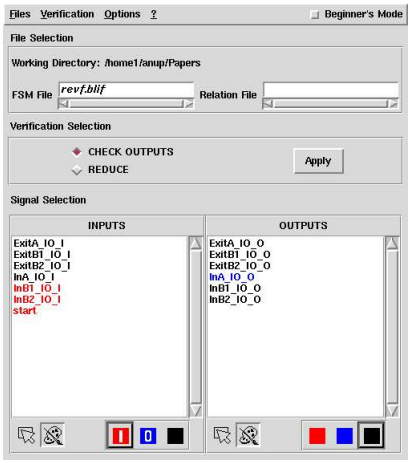


Figure 23: Verification Screen

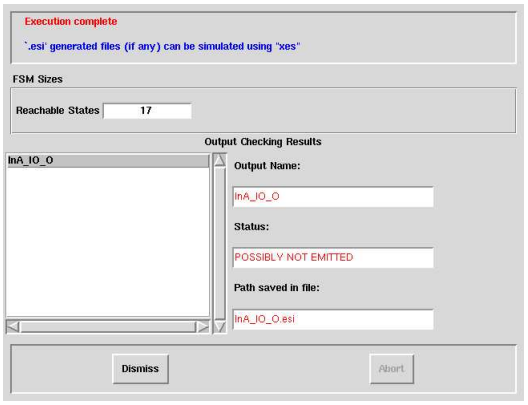


Figure 24: Output of Verification

10 CONCLUSION AND FUTURE WORK

We have explored the specification of operational semantics for the Activity Diagrams of UML 2.0 in a synchronous style. The semantics is good for simulation, code generation and verification. All the constructs can be expressed uniformly in the constructs of Esterel. In this approach the external action done in the activitynode can be easily modeled as an external task in the Esterel language. The exception handling in Petri Nets as shown in [26] is rather difficult which can be modeled easily in our framework. We have later extended the syntax and semantics to handle business processes which are prone to failure and require compensating actions. Further work is required to study capabilities and compare with BPMN [3]. We are also working on verification approaches required for the compensating activities.

REFERENCES

[1] OMG: *Unified Modeling Language : Superstructure*, Version 2.0, Revised Final Adopted Specification, October 8, 2004, Source: WWW.omg.org

[2] Douglass B.P. *Real Time UML Advances in the UML for Real-Time Systems*, Pearson Edition,2004

[3] *Business Process Modeling Notation (BPMN) Specification*, Final Adopted Specification, dtc/06-02-01, available from http://www.bpmn.org

[4] *Business Process Execution Language for Web Services* Ver 1.1, available from http://www-128.ibm.com/developerworks/webservices/library/ws-bpel

[5] *From UML to BPEL, Model Driven Architecture in a Web services world* available from http://www-128.ibm.com/developerworks/webservices/library/ws-uml2bpel

[6] van der Aalst,W. ter Hofstede A.Kiepuszewski B., Barros A., *Workflow patterns*, Distributed and Parallel Databases, 14(3), 2003

[7] Russel N., van der Aalst , W. ter Hofstede A ., Peta Wohed*On the Suitability of UML 2.0 Activity Diagrams for Business Process Modelling*,Proceedings of the Third Asia-Pacific Conference on Conceptual Modelling ,APCCM, 2006.

[8] Yeong W.L. *CSP-Based Verification for Web Service Orchestration and Choreography*Simulation,Society for Computer Simulation International, 2007

[9] Peter Y. H. Wong and Jeremy Gibbons*A Process-Algebraic Approach to Workflow Specification and Refinement*, In Proceedings of 6th International Symposium on Software Composition, March 2007

[10] Brookes S.D. , Hoare C.A.R, Roscoe *A Theory of Communicating Sequential Process*, JACM, Vol 31, 1984

[11] Milner R., *Communication and Concurrency*, Prentice Hall, 1989.

[12] Hoare T. *Compensable Transactions* from Slides Presented atUNI-IIST, Beijing, May 2006

[13] Berry G, Gonthier G., *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*, Science of Computer Programming, 1992.

[14] Butler M., Ferriera C., *A Process Compensation Language*, IFM' 2000, LNCS 1945, 2000

[15] Butler, M., Hoare, C. A. R. and Ferreira, C.,*A trace semantics for long-running transactions.* In Proceedings of 25 Years of CSP, LNCS 3525

[16] Bhattacharjee A.K., Dhodapkar S.D., Seshia S., Shyamasundar R.K. *PERTS: an environment for specification and verification of reactive systems* , Reliability Engineering & Systems Safety Journal, 71(2001), Elsevier, UK, 2001

[17] Rajan B. and Shyamasundar R.K., *An Implementation of Communicating Reactive Processes* IASTED - PDCN'97, Int. Conf. on Parallel and Distributed Computing and Networks, Singapore, 1997

[18] Berry G., Ramesh S., Shyamasundar R.K. :*Communicating Reactive Processes*, 20th ACM Symposium on Principles of Programming Languages, 1993

[19] Magee and Kramer *Concurrency : State Models and Java Programs*, Wiley 1999

[20] Fu X., Bultan T., Su J. *Analysis of interacting BPEL web services*, Proceedings of the 13th international conference on World Wide Web, ACM Press, 2004

[21] Howard Foster, Sebastian Uchitel, Jeff Magee, Jeff Kramer, *Model-based Verification of Web Service Compositions*,Eighteenth IEEE International Conference on Automated Software Engineering (ASE), Montreal, Canada, 2003.

[22] Augusto, J. C., Leuschel, M., Butler, M. and Ferreira, C. *Using the Extensible Model Checker XTL to Verify StAC Business Specifications*. In Proceedings of 3rd Workshop on Automated Verification of Critical Systems (AVoCS 2003)

[23] Mariya Koshkina, Franck van Breugel, *Modelling and verifying web service orchestration by means of the concurrency workbench* ACM SIGSOFT Software Engineering Notes, Volume 29 , Issue 5, September 2004.

[24] Eshuis Rik, *Semantics and Verification of Activity Charts*, Ph.D Thesis, University of Twente, 2002

[25] Bruni R., Melgratti H, Montanari U., *Theoretical Foundations for Compensations in Flow Composition Languages*,POPL 2005

[26] Harald Storrle, *Semantics of UML 2.0 Activities*,German Software Engineering Conference, 2005.

[27] Bhattacharjee A.K., Shyamasundar R. K.: *Validated Code Generation for Activity Diagrams*,Distributed Computing and Internet Technology, Second International Conference, ICDCIT 2005, Bhubaneswar, India, December 22-24, 2005, Proceedings. Lecture Notes in Computer Science 3816 Springer 2005

[28] Emmi M., Majumdar R. *Verifying compensating transactions.*, VMCAI 2007, LNCS 4349.

[29] Sebastian Pavel, Jacques Noye, Pascal Poizat, Jean-Claude Royer,*Java Implementation of a Component Model with Explicit Symbolic Protocols*, Software Composition, 4th International Workshop,2005,Edinburgh, LNCS 3628.

[30] *BPELJ:BPEL for Java technology* available from http://www-128.ibm.com/developerworks/ library/specification/ws-bpelj.

[31] Harel D. and Naamad A., *The STATEMATE Semantics of Statecharts*, ACM Trans. on Software Engineering Method. 5:4 October 1996.

[32] Maraninchi F. and Reymond Y.,. Mode-automata: About modes and states for reactive systems. In European Symposium on Programming, ESOP'98, Lisbon, April 1998

A BRIEF DISCUSSION ON CSP

In CSP the ultimate unit in the behaviour of a process is an event (conditions) which are regarded as instantaneous and A is the set of all events. The behaviour of a process upto some instant of time can be a record of events in which it has participated. The basic CSP processes are

$$P ::= STOP \mid SKIP \mid e \rightarrow P \mid c?x \rightarrow P \mid$$

$$P \Box Q \mid P \sqcap Q \mid P;\ Q \mid P \parallel_x Q \mid P \backslash A \mid \mu x.P(x)$$

The process `STOP` can perform no events: it represents the end of a pattern of behaviour. The process `SKIP` can do nothing but terminate and the future behaviour is determined by the expression following the next sequential composition symbol. The process $e \rightarrow P$ ("e then P") is ready to perform the event e and if this event is performed, the future behaviour of this process is described by term P. The query symbol, ?, denotes a choice of events: the process $c?x \rightarrow P$ is ready to perform any event of the form c.x; if this process performs a particular event c.a, then x takes the value a for the rest of the current scope. The symbol $\Box$ denotes an external choice of behaviours. if x and y are distinct events $(x \rightarrow P \Box y \rightarrow Q)$ describes a process which initially engages in either of the event x or y. The notation $P \sqcap Q$ (P or Q) denotes a process which behaves either like P or like Q, where the choice is made internally and may represent run-time nondeterminism. P;Q denotes a process which initially behaves like P and upon successful termination of P behaves like Q. The parallel composition, $P \parallel Q$, specifies the process which behaves like the system composed of processes P and Q interacting in lock step synchronisation. The set of events that can occur only if performed simultaneously by both processes. In $P \parallel_X Q$, the execution of the activities in P and Q are synchronized over X. The hiding operator internalises sets of events: the expression $P \backslash A$ denotes a process that behaves exactly as P, except that events from the set A are no longer visible in the environment i.e. they may not be shared with, and do not require the cooperation of, other processes. A recursive process like a clock can be defined as $\mu x : (tick) \mid (tick \rightarrow x)$.

Traces play a central role in CSP in describing the behaviour of processes. A trace of the behaviour of a process is a finite sequence of symbols recording the events in which the process has engaged up to some moment in time. $traces[\![\bullet]\!]$ is a semantic function which maps a CSP expression to its set of possible traces. The set of all such traces is defined by $traces[\![P]\!] = \{s \in \mathcal{A}^* \mid \exists Q.P \xrightarrow{s} Q\}$. The general CSP

process composition rules defined in terms of antecedent and consequent $\frac{A_1, A2, \cdots, A_n}{C}$ are defined below :

Termination	$\frac{}{SKIP \xrightarrow{\surd} STOP}$	
Prefix	$\frac{}{a \rightarrow P \xrightarrow{a} P}$	
Choice	$\frac{P \xrightarrow{a} P'}{P \Box Q \xrightarrow{a} P'}$	$\frac{Q \xrightarrow{a} Q'}{P \Box Q \xrightarrow{a} Q'}$
Sequence	$\frac{P \xrightarrow{a} P' \wedge a \neq \surd}{P;\, Q \xrightarrow{a} P';\, Q}$	$\frac{P \xrightarrow{\surd} 0 \wedge Q \xrightarrow{a} Q'}{P;\, Q \xrightarrow{a} Q'}$
Interleaving Parallel	$\frac{P \xrightarrow{a} P' \wedge a \notin \mathcal{A}}{P \|_{\mathcal{A}} Q \xrightarrow{a} P' \|_{\mathcal{A}} Q}$	$\frac{Q \xrightarrow{a} Q' \wedge a \notin \mathcal{A}}{P \|_{\mathcal{A}} Q \xrightarrow{a} P \|_{\mathcal{A}} Q'}$
Synch. Rule	$\frac{P \xrightarrow{a} P' \wedge Q \xrightarrow{a} Q' \wedge a \in \mathcal{A}}{P \|_{\mathcal{A}} Q \xrightarrow{a} P' \|_{\mathcal{A}} Q'}$	

B ESTEREL

The basic object of Esterel without value passing, referred to as PURE Esterel, is the signal. Signals are used for communication with the environment as well as for internal communication. The programming unit is the module. A module has an interface that defines its input and output signals and a body that is an executable statement:

```
module M:
   input I1, I2;
   output 01, 02;
   input relations
   statement
end module
```

At execution time, a module is activated by repeatedly giving it an input event consisting of a possibly empty set of input signals assumed to be present and satisfying the input relations. The module reacts by executing its body and outputs the emitted output signals. We assume that the reaction is instantaneous or perfectly synchronous in the sense that the outputs are produced in no time. Hence, all necessary computations are also done in no time. The only statements that consume time are the ones explicitly requested to do so. The reaction is also required to be deterministic: for any state of the program and any input event, there is exactly one possible output event. In perfectly synchronous languages, a *reaction* is also

called an *instant*. Instantiation of a module is done through the `run` statement. For instance, `run` *exchange [X1/E1, ... Xn/En]* copies the body of the module *exchange* in place of the `run` command after renaming all occurrences of the signals X1, ... Xn by E1, ... En respectively; in other words, the parameters are *bound by capture*.

Asynchronous tasks are those tasks which do take time; that is, the time between initiation and completion is observable. In the terminology of Esterel, this can be interpreted to mean that there will be at least one instant between initiation and completion. The `exec` primitive provides the interface between Esterel modules and asynchronous tasks. An asynchronous task is declared by the statement "`task` task_id (f_par_lst) `return` signal_nm (type);" where `task_id` is the name of the task, `f_par_lst` gives the list of *formal* parameters (reference or value) and the signal returned by the task is given by the `signal_nm` with its type after the keyword `return` Instantiation of the task is done through the primitive `exec`. For example, the above task can be instantiated from an Esterel program as "`exec task_id (a_par_lst);`".

A typical task declaration appears as "`task ROBOT_move (ip, fp) return complete`" and the call appears as "`exec ROBOT_move (x,y)`". The execution of this statement in some process starts task `ROBOT_move` and awaits for the return signal `complete` for it to proceed further. In other words, `exec` requests the environment to start the task and then waits for the return signal.

ABOUT THE AUTHORS

Anup Kumar Bhattacharjee has a Master of Technology from IIT Kharagpur in Computer Science and he is a doctoral student under Prof. R.K. Shyamasundar, TIFR. He is employed with BARC, Mumbai, India. He can be reached at anup@barc.gov.in.

R.K. Shyamasundar has a Ph.D in Computer Science and Automation from IISc, Bangalore. He did his post-doctoral work during 1978-1979 as an International Research Fellow at Eindhoven Technological University, Eindhoven, Netherlands under the famed Professor Dr. Edsgar W Dijkstra. He was the first Dean of the School of Technology and Computer Science at the Tata Institute of Fundamental Research. He had various assignments at IBM TJ Watson Research center, Eindhoven University of Technology, The Netherlands, State University of Utrecht, the Netherlands, Pennsylvania State University, University of Illinois at Urbana, University of California, San Diego, ENSMP Sophia Antipolis, IRISA, Rennes, Verimag Grenoble Max Planck Institute for Computer Science at Saarbrucken etc. He has more than 200 publications and several patents in US and India. Thirty students have done their Ph.D. under his guidance in India and abroad. He can be reached at shyam@tcs.tifr.res.in.