Validated Code Generation for Activity Diagrams

A.K. Bhattacharjee[1] and R.K. Shyamasundar[2]

[1] Reactor Control Division, Bhabha Atomic Research Centre, Mumbai 400 085
anup@barc.ernet.in
[2] School of Technology and Computer Science, Tata Institute of Fundamental Research, Mumbai 400 005
shyam@tifr.res.in

Abstract. *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 synchronous concepts of activity diagrams. The latter can be effectively utilized for validated code generation and verification. In this 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 mechanisms of synchronous language environments such as Esterel and its programming environments. Further, the framework leads to scalable verification methods.

1 Introduction

Activity Diagram is one of the important diagrams in UML. 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. In UML 2.0, activity diagrams support concurrent control and data flow, loops, conditionals and exception handling. The two basic entities are *Actions* and *Activities*. An *Action* is the fundamental unit of executable functionality and an *activity* provides the coordinated sequencing of subordinate units whose individual elements are actions. This coordination is expressed as a graph of *ActivityNodes* connected by *ActivityEdges*. Since there are actions that invoke activities, that may be nested and possibly form invocation hierarchies invoking other activities (ultimately resolving to individual atomic actions). The OMG document [6] classifies activity diagrams as *Fundamental, Basic, Intermediate, Structured, Complete* in terms of complexity in the process flow. In this paper, we are concerned with the Intermediate Level of Activity Diagrams that include control and data flow and decisions. A simple activity diagram describing the order processing and account is shown in Fig. 1.

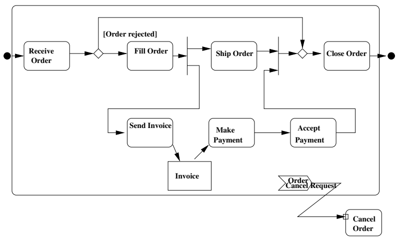


Fig. 1. Simple Activity Diagram

Although the OMG document [6] 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.

Most of the works on the semantics of UML activities in general have been based on Petri nets. Two of the significant efforts toward formalization of UML activities are [7] and [8]. Eshuis [7] 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. It is to be noted that the implementation semantics has not been obtained as a refinement of the requirement level semantics. The semantics also covers activity charts of UML 1.5 but not of activity diagrams of UML 2.0[1]. Storrle [8] 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.

In this paper, we propose a reactive formalism of Activity Diagrams of UML 2.0 description; for description purpose we use Esterel language. 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 activity diagrams but also generate validated code automatically.

2 Activity Diagrams: Informal Interpretation

An action is the fundamental unit of executable functionality in an activity [6]. The execution of an action represents some transformation or processing in the

[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.

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 flow edges within activities, which carry control and object tokens respectively. An action can only begin execution when all incoming control tokens are present and all input pins have object tokens. 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 tokens in its outgoing control edges and output pins, where they are accessible to other actions.

3 Synchronous Framework for Activity Diagrams

In this section, we capture activity diagrams 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 below.

3.1 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 O1, O2;
   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.

4 Synchronous Interpretation of Basic Activity Diagrams

The synchronous model for the Activity Diagrams is represented as a collection of transformation rules for each construct of the Activity Diagrams. 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. 2, the module `simpleActivity` performs the task of passing control tokens from the activity `sendPayment` to the activity `receivePayment` and is the the root activity. The activities `sendPayment`, `receivePayment` and `simpleActivity` in the above example, can be interpreted through the Esterel fragments shown in the Fig.2.

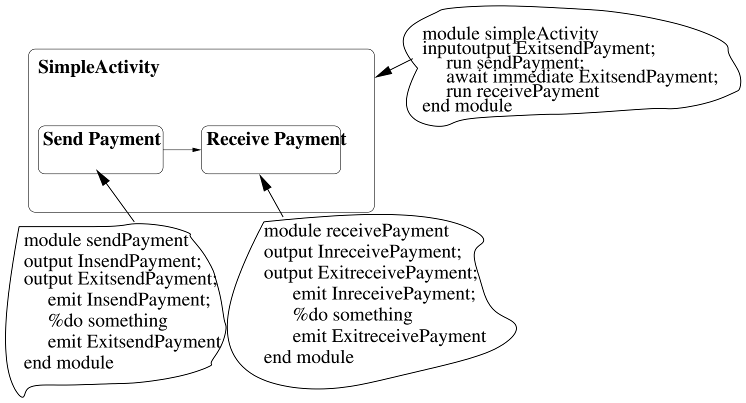


Fig. 2. Simple node

Merge Node: A merge node (cf. Fig. 3) is a control node that brings together multiple alternate flows. It is not used to synchronize concurrent flows but to accept one among 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
```

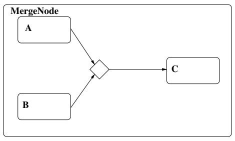


Fig. 3. Merge Node

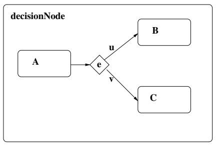


Fig. 4. Decision Node

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 the recent OMG document [6].

Decision Node: A decision node (cf. Fig. 4) is a control node that chooses between the outgoing flows. It has one incoming edge and multiple outgoing edges. It can be described by the following Esterel fragment.

```
module decisionNode
var e in
       run A;
       if e = u
          run B; % e is the guard which if  u then run B
       else if e = v
          run C; % e is the guard which if  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. 5) 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.

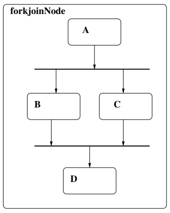


Fig. 5. Fork Join Node

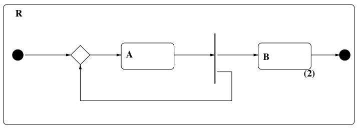


Fig. 6. Reentrant Node

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

```
module forkJoinNode

    run A       % run activity A
       [
         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. 6. 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. 6 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.6 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] % Signal renaming
        ]
        else [ present not InB2 then
               [                 % Second instance of B
                await ExitA;
                emit b1b2;
                run B2/B[signal ExitB2/ExitB,InB2/InB] %Signal renaming
           ]
               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. 7, 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. 7, Activity Node

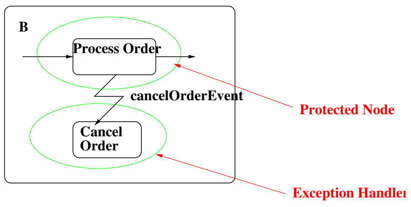


Fig. 7. Exception Node

`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
              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`.

4.1 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.8. 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.

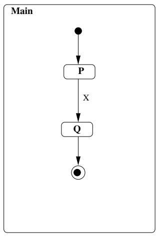


Fig. 8. Object node with data

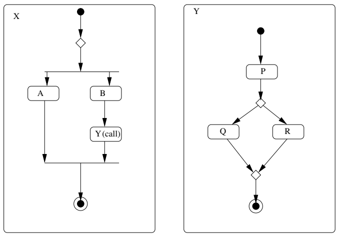


Fig. 9. Activity with Nesting

```
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
...
    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. 9. This can be modeled by using the *run Y* construct of Esterel. The following Esterel fragment describes the nested call of the Fig.9.

```
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
```

4.2 Communication in Activity Diagrams

The notion of communication between two Activity Diagrams can be nicely modeled in the Communicating Reactive Processes (CRP) [3] framework. The CRP model consists of network $M1||M2||..Mn$ 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 [3]. 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. 10, shows a simple example of an activity diagram showning 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 [1] 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 [4] 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.

5.1 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

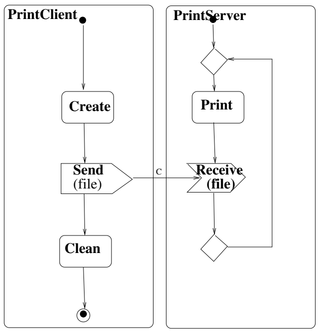


Fig. 10. Activities with communication

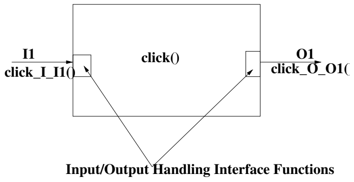


Fig. 11. Activity to Code Mapping

by construction, using Esterel-C/Java code generators. We assume Esterel-C code generator for further discussion.

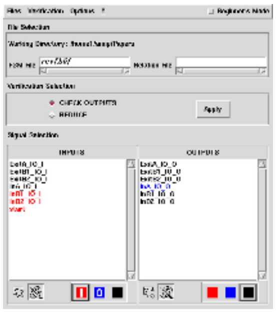
For actual execution of the code , the generated code must also be linked with some extra layer of code that realizes the interface with the outside world which detects input events, read data and realizes output events and send data.If for example the module `click` should react to an input event, composed for example of one input tokens I1 as shown in Fig. 11. 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 [4, 5].
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 Verification

The above 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



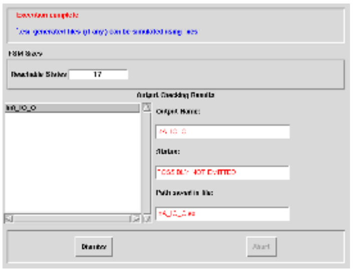
Working Directory: [illegible]

FSM file: [illegible]

[✓ CHECK OUTPUTS] REDUCE

INPUTS	OUTPUTS
ExtA_IO_I	ExtA_IO_O
ExtB1_IO_I	ExtB1_IO_O
ExtB2_IO_I	ExtB2_IO_O
InA_IO_I	InA_IO_O
InB1_IO_I	InB1_IO_O
InB2_IO_I	InB2_IO_O
start	

Fig. 12. Verification Screen



Execution complete

[illegible] generated files (if any) can be simulated using [illegible]

Reachable States: 17

InA_IO_O

Output Name: InA_IO_O

Status: POSSIBLY NOT EMITTED

Path saved in file: [illegible]

Fig. 13. Output of Verification

model can then 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. 6 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.12,13 for reference.

7 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. Our initial experience shows that verification of Activity Diagrams in this approach can be applied to moderately large examples. Further study is in progress. 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 [8] is rather difficult which can be modeled easily in our framework. Presently, we are building a translator which can translate the activity diagrams into Esterel models. We need to test the effectiveness of the Esterel code generators in the context of real-life activity diagrams.

Acknowledgment

The first author wishes to acknowledge the travel grant made by DST, Government of India under the ITPAR proposal for visiting the University of Trento, Italy during which most part of the work was carried out. The first author also wishes to acknowledge the encouragement and institutional support received from Mr. S.D. Dhodapkar, Head, Software Reliability Section, BARC during the work, without which this would not have been possible.

References

1. Rajan Basant and Shyamasundar R.K., *An Implementation of Communicating Reactive Processes* IASTED - PDCN'97, Int. Conf. on Parallel and Distributed Computing and Networks, Singapore, 1997
2. Berry G, Gonthier G., *The Esterel synchronous programming languages: Design, semantics,implementation.*, Science of Computer Programming, 19(2):87-152, 1992
3. Berry G., Ramesh S., Shyamasundar R.K. :*Communicating Reactive Processes*, 20th ACM Symposium on Principles of Programming Languages, 1993
4. 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.
5. Seshia S., Shyamasundar R.K., Bhattacharjee A.K., Dhodapkar S.D. *A Translation of Statecharts to Esterel* Lecture Notes in Computer Science, Vol 1698, Springer, 1999
6. OMG: *Unified Modeling Language : Superstructure*, Version 2.0, Revised Final Adopted Specification, October 8, 2004, Source: www.omg.org
7. Eshuis Rik, *Semantics and Verification of Activity Charts*, Ph.D Thesis, University of Twente, 2002
8. Harald Storrle, *Semantics of UML 2.0 Activities*,German Software Engineering Conference, 2005.