Semantics and Verification of Hierarchical CRP Programs*

R.K. Shyamasundar
Computer Science Group
Tata Institute of Fundamental Research
Bombay 400 005, India
e-mail: shyam@turing.tifr.res.in
S. Ramesh
Department of Computer Science and Engineering
Indian Institute of Technology
Bombay 400 076, India
e-mail: ramesh@cse.iitb.ernet.in

Abstract

Communicating Reactive Processes (CRP) paradigm unifies asynchronous and synchronous mechanisms of concurrent programming languages. As an example, CRP unifies Esterel and Hoare's Communicating Sequential Processes (CSP). It has been earlier shown that such a unification and in particular CRP can be used for the specification of hybrid systems and dynamic real-time systems. In this paper, we extend the CRP to support hierarchical refinement and describe a compositional semantics. Further, we show how verification can be done using the verification techniques and tools used for the verification of Esterel programs. We illustrate how a careful separation of Esterel and CSP mechanisms in CRP has enabled the use of the technique of *verification by reductions* for verification and illustrate the use of tools such as Auto/Autograph for the verification of CRP programs through the banker-teller example.

1 Introduction

Programming languages that have been used for real-time programming can be broadly categorized into:

1. *Asynchronous Languages*: Here, a program is treated as a set of loosely coupled independent execution units or processes, each process evolving at its own pace. Interprocess communication is done by mechanisms such as message passing. Communication as a whole is *asynchronous* in the sense that *an arbitrary amount of time can pass between the desire of communication and its actual completion.* This class includes languages such as Ada, Occam, CSP etc.

* The work was partially supported by IFCPAR (Indo-French Centre for the Promotion of Advanced Research), New Delhi, India.

2. *Perfectly Synchronous Languages*: In this class, programs react instantaneously to its inputs by producing the required outputs. Statements evolve in a tightly coupled input-driven way deterministically and communication is done by instantaneous broadcast where the receiver receives a message exactly at the time it is sent. That is, a perfectly synchronous program produces its outputs from its input with *no observable time delay*. Languages such as Esterel [2], Lustre, Signal, Statecharts belong to this category. These languages are often referred to as *reactive* languages. These languages use the multiform notion of time rather than clock time.

Programming languages based on the *perfect synchrony paradigm* have proven useful for programming reactive systems where determinism is a must and logical concurrency is required for good programming style. One of the main reasons for its success is due to the fact that it permits the programmer to focus on the logic of reactions and makes it possible to use several automata-based verification systems for correctness proofs. Further, the correctness proofs of programs follow their implementation very closely and hence, are more robust and reliable. On the other hand, asynchronous languages have proven useful in distributed processing and implementing algorithms on a network of computer systems where nondeterminism is an appropriate abstraction at the logical and physical levels. Presently, each class of languages is unable to handle problems to which the other class is tailored. Asynchronous languages are inappropriate for truly reactive systems that require deterministic synchronous (perfect) communication. On the other hand, existing synchronous languages lack support for asynchronous distributed algorithms. However, complex systems do require the abilities of both the languages. For instance, a robot driver must use a specific reactive program to control each articulation, but the global robot control may be necessarily asynchronous because of limitations of networking capabilities. General specification of complex reactive systems needs capabilities such as:

1. Relating asynchronous events (events that can happen arbitrarily close to each other),
2. Integration of discrete and continuous components (continuous components may cause continuous change in the values of some state variables according to some physical law), and
3. Referencing explicit clock times.

In [4], a paradigm referred to as *Communicating Reactive Processes* (CRP), has been proposed that provides a unification of perfect synchrony and asynchrony. CRP permits only top-level network parallelism. In [7] an extension of CRP, referred to as *Timed* CRP has been proposed having the capabilities to specify hybrid and asynchronous dense-time real-time systems and in [8], we have investigated the power of the CRP paradigm in providing a platform for programming dynamic real-time systems.

In this paper, we extend CRP to support hierarchical refinement. We show how the techniques for describing a compositional semantics of ESTEREL envisaged in [11], can be generalized for hierarchical CRP programs. The compositional behavioral semantics makes it possible to use techniques of ESTEREL verification for the verification of hierarchical CRP programs. Essentially, the compositional semantics allows us to split the proof into verification of ESTEREL nodes and the ESTEREL network. Thus, transformations such as behaviour abstraction, filtering and hiding can be applied for the nodes and for the network as well. We illustrate the verification of a CRP program using the above transformations using Auto/Autograph. In other words, we establish that proving CRP programs is no harder than proving ESTEREL programs. That is, the unification of perfect synchrony and asynchrony does not complicate the proof techniques. We illustrate the technique through the Banker-Teller example given in [4].

2 Hierarchical Communicating Sequential Processes

First, we give an informal introduction and rationale for CRP before describing the syntax and behavioral semantics.

2.1 Communicating Reactive Processes

A CRP program consists of independent locally reactive ESTEREL nodes that communicate with each other by rendezvous communication as in Hoare's Communicating Sequential Processes (CSP). CRP faithfully extends both ESTEREL and CSP and adds new possibilities such as precise local watchdogs on rendezvous.

A CRP program consists of a network $M_1 \; // \; M_2 \; \cdots \; // \; M_n$ of ESTEREL reactive programs or *nodes*, each having its own input/output reactive signals and its own notion of an instant. The network is asynchronous (in the sense discussed earlier) and each node M_i is locally reactive driving a part of a complex network that is handled globally by the network.

Basic Interface to Asynchrony

In full ESTEREL , external procedure calls are assumed to be instantaneous. Such a feature is not convenient for dealing with actions which are known to take time – for instance, even (long!) numerical computation required in many situations are known to take time. The new `exec` statement overcomes this draw back and provides a gateway to asynchrony. An asynchronous task is declared by "**task P**" (the precise syntax is given in the sequel) and task execution is controlled from ESTEREL by the statement

exec P

that waits for the task to terminate. Since there can be several occurrences of "**exec P**" in a module for the same task P, several simultaneously active tasks having the same name can coexist. In fact, to overcome the ambiguity, we permit the labelling of the `exec` statement (explicitly and implicitly).

Basic Rendezvous Statements

The central idea of establishing asynchronous communication between nodes lies in extending the basic `exec` primitive into a communication primitive. The usual `send` and `receive` asynchronous operations can be represented by particular tasks that handle the communication. Several `send/receive` interactions are possible according to various types of asynchronous communication. For instance, `send` can be non-blocking for full asynchrony, or `send` and `receive` can synchronize for CSP-like rendezvous communication. All choices can be implemented through ESTEREL . In this paper, we concern ourselves with the integration of ESTEREL with CSP and consider the rendezvous communication between nodes.

CRP nodes are linked by *channels*. We start by describing the pure synchronization case, where channels are symmetric. Pure channels are declared in CRP nodes by the declaration

```
channel C
```

A channel must be shared by at most two nodes. Communication over the channels is handled by the statement

```
rendezvous L:C
```

The `rendezvous` statement is a particular instance of `exec` statement and hence, the interface detailed above follows. A given channel can only perform one rendezvous at a time. Hence, in each node, the return signals of all the `rendezvous` statements on the same channel are *implicitly assumed (declared) to be incompatible*[2]. Values can be passed through unidirectional channels. In the following section, we describe the syntax of CRP and the declaration and calling conventions of asynchronous tasks.

2.2 Syntax of the Hierarchical CRP

In the following, we describe the abstract syntax of hierarchical CRP programs. We will not describe the syntax of ESTEREL statements for the sake of brevity; we will give the structure of kernel statements only.

```
< crp_prog >   ::=    < stat >  |   < network >
                   (* The usual ESTEREL kernel statements *)
< stat >   ::=   nothing
                 | halt
                 | emit S
                 | stat1; stat2
                 | loop stat end
                 | present S then stat1 else stat2 end
                 | do stat watching S
                 | stat1 || stat2
                 | trap T in stat end
                 | exit T
                 | signal S in stat end

                   (* Additional Statements in CRP *)
```

[2] This is needed for enforcing CSP restrictions only.

```
                (* task declaration *)
  | task task_id (par_lst)  return signal_nm (type)
                (* channel declaration *)
  | channel ch_name (type)
                (* instantiating a task *)
  | exec task_id (par_lst) return signal_name

                (* sending a value on the channel *)
  | rendezvous_send channel_id channel_name
      (expression) return signal_name (some_type)

                (* sending a value on the channel *)
  | rendezvous_receive channel_id channel_name
      (variable) return signal_name (some_type)

                 (* Hiding Statement *)

  | [< network >]
```

$< network >$ ::= $process_1 \;\; || \; process_2$

$< process >$ ::= `process begin` $< stat >$ `end` | $< network >$

The kernel statements are imperative in nature, and most of them are classical in appearance. The `trap-exit` constructs form an exception mechanism fully compatible with parallelism. Traps are lexically scoped. The local signal declaration "`signal` S `in` *stat* `end`" declares a lexically scoped signal S that can be used for internal broadcast communication within "*stat*". The `then` and `else` parts are optional in a present statement. If omitted, they are understood to be the same as `nothing`.

Task/channel declarations and the basic asynchronous and rendezvous executions are described below.

The Hierarchical Parallel Operator

In CRP [4], the network parallel operator can be used only at the top level. In the hierarchical CRP described in this paper, the network parallel operator can appear anywhere in the program text. This generality also permits us to introduce the analogue of "hiding" in CRP – thus, permitting hierarchical abstraction. We enforce the following discipline on the tasks with reference to sharing of signals and variables:

- Processes on different nodes do not share variables.
- Processes on different nodes can share only pure input signals.
- Any two processes denoting nodes can share common channel signals.

Task Declaration

An asynchronous task is declared as follows:

```
task task_id (par_lst)  return signal_nm (type);
```
where

- `par_lst` gives the list of parameters (reference or value);
- the signal returned by the task is given by the `signal_nm` with its type after the keyword `return`; the return signal is optional and it is possible to have multiple return signals.

A typical example of the declaration of a component is given by:

```
task MOUSE (y) return Mouse ({sanct,kill})
```

Channel Declaration
Channels are declared in CRP nodes by the declaration:

```
channel C (type)
```

A Channel is shared exactly by two nodes and the declarations should type-match.

Task Instantiation
Instantiation of the task is done through the primitive `exec`. For example, the above task can be instantiated from an ESTEREL program as follows:

```
exec MOUSE (y) return {win,loss,tie}
```

The execution of the above statement starts task `MOUSE` and waits for its termination; of course, the number and type of arguments, and the return signal type should match with the task declared. In other words, `exec` requests the environment to start the task and then waits for the return signal (which also indicates the termination of the task).

Since there can be several occurrences of `exec T` in a module for the same task `T`, several simultaneously active tasks having the same name can coexist. To avoid confusion among them one can assign an explicit label to each `exec` statement; a labeled `exec` statement is of the form `exec L:T`. The label name must be distinct from all other labels and input signal names. An implicit distinct label is given to `exec` statements.

Operational View of `exec`
The primitive `exec` provides an interface between ESTEREL and the asynchronous environment can be seen by the following interpretation. Given an `exec` statement labeled L, the asynchronous task execution is controlled from ESTEREL by three implicit signals `sL` ((output signal)`,` `L` (input signal)`,` and `kL` (output signal) corresponding to starting the task, completion and killing the execution of the task respectively. The output *start* signal `sL` is sent to the environment when the `exec` statement starts. It requests the start of an asynchronous incarnation of the task. The input *return signal* `L`, is sent by the environment when the task incarnation is terminated; it provokes instantaneous termination of the `exec` statement. the output *kill signal* `kL` is emitted by ESTEREL if the `exec` statement is preempted before termination, either by an enclosing watching statement or by concurrent exit from an enclosing trap. The return signals corresponding to the `exec` label can be used for declaring incompatibility with other input signals (this becomes handy in declaring channels).

Rendezvous Instantiation

The rendezvous in the sender is initiated through the command:

```
rendezvous_send channel_name (value_expression) return (some_type)
```

The rendezvous in the receiver is initiated through the command:

```
rendezvous_receive channel_name (variable) return (some_type)
```

The other explanations follow on the same lines as that of the task instantiation given above.

For convenience[3], we use the following construct for guard selection in CRP:

```
rend_case
    case  L1: rendezvous_receive channel1 (x) do stat_1
    case  L2: rendezvous_send channel2 (exp) do stat_2
end
```

The code corresponds to selecting a *receive* on `channel1` followed by the execution of the statement-sequence *stat1* or *send* on `channel2` followed by the execution of the statement-sequence *stat2*.

2.3 Behavioural Semantics

In the classical ESTEREL, given a module M and an input event I, the behavioral semantics determines a transition $M \xrightarrow[I]{O} M'$ where O is the generated output event and M', the derivative is another module suited to perform further reactions. In this reaction, the total event consists of all the input and output signals. The derivative M' has the same interface as M and differs by its body. The reaction to a sequence of input events is computed by chaining such elementary transitions. In [11], a compositional semantics of ESTEREL has been described with the purpose of adapting the technique of *verification by reductions* for verifying ESTEREL programs. The primary distinction of the transition systems generated lies in the structure of action labels. In the semantics, the action labels are triples consisting of: (i) necessarily present signals, (ii) necessarily absent signals and (iii) signals emitted during the reaction. In [11], a relation among the input and output signals during each instant is used for checking causality. However, this is no longer needed due to the causality schemes proposed recently by G. Berry [3] and Olivier Ploton [6]. In the context of CRP, we have to keep track of pending asynchronous tasking requests and do the bookkeeping of the successful completion of the task or preemption of the task. In the semantics of CRP described in [4], we used an action structure consisting of (i) the input signals present, (ii) the output signals emitted, (iii) the trap set or the termination levels and (iv) the set of pending asynchronous tasks. The bookkeeping for generating the appropriate kill signals was done through a syntactic decoration of the *present and trap* constructs. In this paper, we describe a compositional semantics of the hierarchical CRP without resorting to such ad-hoc decorations. It can be easily seen that "the syntactic decorations" are in a sense state-information. For this reason, in our approach we keep track of an explicit information with reference

[3] It may be noted that nondeterministic guard selection can be done in CRP without any additional operator; cf. [4].

to preemptive tasks and signals separately as an environment. Thus, the transition system takes the form,

$$< stat, environment > \xrightarrow{<A>,L} < stat', environment' >$$

where:

Environment: Environment reflects the signals that can kill the rendezvous/asynchronous tasks. In other words, the environment, keeps track of the signals that can preempt (kill) asynchronous tasks or the rendezvous. Formally, the environment (usually denoted by η), is denoted by $\{< L_i, K_i >\}$ where L_i is an active rendezvous/asynchronous_task label which can be preempted by any of the set of signals in K_i. Note that K_i would have at least the signal kL_i which is the signal generated by the Esterel node for killing the task labelled L_i. Note that for killing a rendezvous/task labelled, L, it is necessary that a kill signal kL is generated in the Esterel node.

Action: The action label[4] is a tuple $< E >, L$ where

- E is the set of signals consisting of
  - The necessarily present signals, I^+,
  - The necessarily absent signals, J^-,
  - The output signals, O, emitted and
- L is the set of completed rendezvous labels/asynchronous task labels in the current instant.

Thus, an event is of the form $< I^+, J^-, O >, L$.

It may be noticed that , our action labels will not have the termination levels as we are omitting the traps to begin with. We will generalize the rules later to handle traps.

Esterel permits testing of absence of signals. Thus, we would like to capture the behaviour in terms of an event that is based on a set of signals present and a set of signals that are absent so that it will be possible to compose modules. Due to the synchrony hypothesis, there would be some cycles (or leads to equations which do not have solutions) which have to be eliminated for the computational semantics. Recently the problem of detecting cycles has been completely solved and the algorithms based on the electrical semantics and obtaining causal-free programs complete with respect to the behavioral semantics is reported in [3] and [6] respectively. We will not consider causality aspects in the sequel.

[4] We ignore `traps` at this point.

Notation:

1. We use a predefined signal "fin" to indicate sequential termination; it corresponds to a boolean flag or a general termination level used in the SOS rules in [1, 2]. Note that fin can only be emitted (similar to the *_exit* signal used in [11]).

2. We define operator # used in two ways, one in the context of environment, η, defined above and the other in the context of a set of pending asynchronous tasks, L.

   – Let $E =< I^+, J^-, O >$ be an event, η an environment, and S an input signal. Then, $< E, \eta > \#S = < I^+, J^-, O' >$ where

   $$\begin{aligned} O' = O \cup \{kL| < L, \{S\} >\in \eta, \text{ and } sL \notin O\} \\ -\{sL| < L, \{S\} >\in \eta, \text{ and } sL \in O\} \end{aligned}$$

   That is, $< E, \eta > \#S$ differs from E by the `rendezvous`'s/`exec`'s started before the current instant that are killed by S and that `rendezvous`'s started at the current instant being watched under S are simply ignored or killed right away.

   – Let $E =< I^+, J^-, O >$ be an event, L be a set of rendezvous/exec labels. Then, $E\#L = < I^+, J^-, O' >$ where

   $$\begin{aligned} O' = O \cup \{k\ell|\ell \in L \text{ and } s\ell \notin O\} \\ -\{s\ell|\ell \in L, \text{ and } s\ell \in O\} \end{aligned}$$

3. Let η be an environment and S a signal. Then,

$$\begin{aligned} \eta \uparrow S &= \{< H, \alpha \cup \{S\} > \mid < H, \alpha >\in \eta\} \\ \eta \downarrow S &= \{< H, \alpha \backslash \{S\} > \mid < H, \alpha >\in \eta\} \end{aligned}$$

In the following, we describe the behavioural semantics using rules are of the form

$$< P, \eta > \xrightarrow{E,L} < P', \eta' >$$

For the sake of brevity, we consider pure ESTEREL rules and hence, in the rendezvous we do not distinguish between `rendezvous_send` and `rendezvous_receive`. Hence, for the sake of brevity, we use `rendezvous` for denoting either of them.

Rules for the basic constructs of ESTEREL are given in Table Ia. The interpretation of the rules (*halt*), (*nothing*), (*emit*), (*seq_1*), and (*seq_2*) follows on the lines of the rules for ESTEREL.

halt
$< \texttt{halt}, \emptyset > \xrightarrow{<\emptyset\ ,\emptyset,\emptyset>,\ \emptyset} < \texttt{halt}, \emptyset >$
nothing
$< \texttt{nothing}, \emptyset > \xrightarrow{<\emptyset\ ,\emptyset,\{fin\}>,\ \emptyset} < \texttt{nothing}, \emptyset >$
emit
$< \texttt{emit S}, \emptyset > \xrightarrow{<\emptyset\ ,\emptyset,\{S\}\cup\{fin\}>,\ \emptyset} < \texttt{nothing}, \emptyset >$
Seq_1
$< stat_1, \eta_1 > \xrightarrow{<I_1^+,J_1^-,O_1\cup\{fin\}>,\emptyset} < stat_1', \emptyset >,\ fin \notin O_1$ $< stat_2, \emptyset > \xrightarrow{<I_2^+,J_2^-,O_2>,\emptyset} < stat_2', \eta_2' >,$ $I_1^+ \cap J_2^- = I_2^+ \cap J_1^- = \emptyset,\ O_1 \cap J_2 = O_2 \cap J_1 = \emptyset$ $\overline{< stat_1; stat_2, \eta_1 > \xrightarrow{<I_1^+ \cup I_2^+, J_1^- \cup J_2^-, O_1 \cup O_2>,\emptyset} < stat_2', \eta_2' >}$
Seq_2
$\dfrac{< stat_1, \eta_1 > \xrightarrow{<I_1^+,J_1^-,O_1>,L_1} < stat_1', \eta_1' >, fin \notin O_1}{< stat_1; stat_2, \eta_1 > \xrightarrow{<I_1^+,J_1^-,O_1>,L_1} < stat_1'; stat_2, \eta_1' >}$
Table Ia: Inductive Rules

The rule for the loop-statement is given below:

loop:
$\dfrac{< stat, \eta > \xrightarrow{<I^+,J^-,O>,L} < stat', \eta' >}{< \texttt{loop}\ stat\ \texttt{end}, \eta > \xrightarrow{<I^+,J^-,O>,L} < stat';\ \texttt{loop}\ stat\ \texttt{end}, \eta' >}$

The rules for the `exec` and `rendezvous` are given in Table Ib.

exec_st $< \texttt{exec}\ T, \emptyset > \xrightarrow{<\emptyset\ ,\emptyset,\{sT\}>,\emptyset} < \texttt{exec\_wait}\ T,\ \{< T, \emptyset >\} >$
exec_wt1 $< \texttt{exec\_wait}\ T, \{< T, \emptyset >\} > \xrightarrow{<\emptyset\ ,\emptyset,\emptyset>,\emptyset} < \texttt{exec\_wait}\ T,\ \{< T, \emptyset >\} >$
exec_wt2 $< \texttt{exce\_wait}\ T, \{< T, \emptyset >\} > \xrightarrow{<\emptyset\ ,\emptyset,\{fin\}>,\{T\}} < \texttt{nothing}, \emptyset >$ *rend_st* $\dfrac{H \text{ is a channel}}{rend < \texttt{rendezvous}\ H, \emptyset > \xrightarrow{<\emptyset\ ,\emptyset,\{sH\}>,\emptyset} < \texttt{rend\_wait}\ H,\ \{< H, \emptyset >\} >}$
rend_wt1 $\dfrac{H \text{ is a channel}}{< \texttt{rend\_wait}\ H, \{< H, \emptyset >\} > \xrightarrow{<\emptyset\ ,\emptyset,\emptyset>,\emptyset} < \texttt{rend\_wait}\ H,\ \{< H, \emptyset >\} >}$
rend_wt2 $\dfrac{H \text{ is a channel}}{< \texttt{rend\_wait}\ H, \{< H, \emptyset >\} > \xrightarrow{<\emptyset\ ,\emptyset,\{fin\}>,\{H\}} < \texttt{nothing}, \emptyset >}$
Table Ib: Inductive Rules

The rule *exec_st* is used to start a *asynchronous task* and the statement is rewritten into an auxiliary `exec_wait` statement that does not exist in the language proper but is used for convenience in defining the semantics. The environment on the left-side of the rule *exec_st* is empty as there cannot be any pending task at this point; on the right-side of this rule, the tuple $< T, \emptyset >$ is added reflecting the fact that the asynchronous task labelled T cannot be preempted by any other signal thus far. In rule (*exec_wt1*), nothing happens; that is, the ESTEREL node is waiting for the completion signals from the asynchronous task. In rule (*exec_wt2*), the task has been completed and hence, the tuple $< T, \emptyset >$ is removed from the environment. The rules *rend_st*, *rend_wt1* and rend_wt2 are the same as that of the corresponding *exec* rules except for checking for the channel declaration. In fact, in an isolated way, there is not much of a difference between the `exec` and the *rendezvous* locally; however, they differ in parallel composition (and hence, implementation). The rules *seq_1* and *seq_2* should be obvious except for the environment component. The environment component for $stat_1$ need not be empty due to the recursive application of the rule (note that there can be instantaneous transitions resulting in the addition of the environment); however, the environment for $stat_2$ remains empty as there cannot be any other pending tasks in it unless $stat_1$ terminates.

Rules for Watching Statement:

The rules are described in Table II. We express these rules in terms of the *present* and *watching_immediate* statements as the structure enables us to show the possible environment for the components and the corresponding updating. In the case of rule (*watching_stat*) the environment on the left is empty as there cannot be any pending task to start with and further, the watching-statement is rewritten as a present-statement. Note that the rule adds "S" as a possible preemptive signal for all the pending rendezvous/exec labels. It may be noted that we could have used the *watching* statement instead of the *watching-immediate* statement in this rule; the reason for using the latter is that we would like to add the preemptive signal only once; the addition of the preemptive signal is done only through the *watching* statement; note that *watching immediate* and *present* statements have similarities.
Watching-Immediate and *Present* Statements: Rule (*watching_immed1*) reflects the behaviour when the signal S is not present in the current instant. In this case, the behaviour would be reflected by the transitions of *stat*; note that the environment need not be empty as it could have been obtained through a rewrite. Since the testing takes place immediately, there is no need to add in the environment the preemptive signal. The rule (*watching_immed2*) corresponds to preemption due to the presence of S in the current instant. It preempts all the pending rendezvous/asynchronous tasks.

Rules (*present1*) and (*present2*) correspond to situations where the transitions correspond to the then-clause and the else-part of the present statement. Since the execution corresponding to the then-clause can take place only after verifying the presence of S, the environment for $stat_1$ in rule (*present1*) is empty; in rule (*present2*) the environment for $stat_2$ need not be empty since instantaneous transitions after verifying the absence of S can lead to change in the environment. Note that (*present1*) and (*watching_immed2*) are alike. Actually, rule (*present2*) handles two situations: if the "present-statement" had arisen due to rewrite of the "watching-statement", then it implies that the embedded statement has inherited some environment; other wise, $\eta = \emptyset$, as the statement could not have been fired. In either case, the effect is given by the rule.

The rules for local signal declaration, the network parallel operator, and hiding are given in Table III. It may be noted, that the complete local signal-declarations is captured by rule *Signal_decl*. It is of interest to note this rule completely captures the behaviour since all operators maintain coherency of events, i.e., $S \in J^- \Rightarrow \not(S \in O)$. The operators are explained below.

Parallel Operator

Rules (*Net_Par_Syn*) and (*Net_Par_Int*) correspond to the rules corresponding to synchronization and interleaving actions. The synchronization is expressed with reference to the common set of channels, denoted CC, between the two processes. Note that it is here one can differentiate an *exec* and a *rendezvous*.

Nesting of Processes

The semantics of process essentially depends on the semantics of a "process", sharing of signals, variables and termination aspects. We impose the following conventions on the programs:

1. Variables are not shared as in the case of ESTEREL modules,
2. Nested processes can share only ESTEREL pure input signals.
3. When a process is preempted all its children get preempted.

Since processes can also be killed, we enrich the environment to consist of tuples of the form

- $< process, set\_of\_kill\_signals >$ in addition to the tuple
- $< rendezvous\ labels,\ set\_of\_kill\_signals >$

with the usual interpretation.

Rule (*hiding_initiate_1*) is similar to the semantics of a run-module of ESTEREL or the rendezvous of CRP. It rewrites into a process or module with the addition of possible process preemption signals (which are empty at this point). Rules (*hiding_ext1*), (*hiding_ext2*) and (*hiding_ext3*) correspond to the network that has neither finished nor preempted, properly finished and preempted respectively. In all these rules, if L is empty, then the action is treated as an internal action. In rule (*hiding_ext2*), we have used an auxiliary statement end_P_1 in place of `nothing`; this rewriting will be helpful if we want to synchronize with external clocks. Rule (*hiding_ext3*) corresponds to preemption of the network. As usual we have used priority for the `kill` signal.

est_Par1

$$\frac{\begin{array}{l} < stat_1, \eta_1 > \xrightarrow{<I_1^+, J_1^-, O_1>, L_1} < stat'_1, \eta'_1 > \\ < stat_2, \eta_2 > \xrightarrow{<I_2^+, J_2^-, O_2>, L_2} < stat'_2, \eta'_2 >, \\ I_1^+ \cap J_2^- = I_2^+ \cap J_1^- = \emptyset, O_1 \cap J_2^- = O_2 \cap J_1^- = \emptyset,\ fin \notin O_1 \cap O_2 \end{array}}{\begin{array}{l} < stat_1 || stat_2, \eta_1 \cup \eta_2 > \xrightarrow{<I_1^+ \cup I_2^+, J_1^- \cup J_2^-, O_1 \cup O_2 - \{fin\}>, L_1 \cup L_2} \\ \qquad\qquad < stat'_1 || stat'_2, \eta'_1 \cup \eta'_2 > \end{array}}$$

est_Par2

$$\frac{\begin{array}{l} < stat_1, \eta_1 > \xrightarrow{<I_1^+, J_1^-, O_1>, L_1} < nothing, \emptyset >,\ fin \in O_1, \\ < stat_2, \eta_2 > \xrightarrow{<I_2^+, J_2^-, O_2>, L_2} < nothing, \emptyset >,\ fin \in O_2, \\ I_1^+ \cap J_2^- = I_2^+ \cap J_1^- = \emptyset,\ O_1 \cap J_2 = O_2 \cap J_1 = \emptyset, \end{array}}{< stat_1 || stat_2, \eta_1 \cup \eta_2 > \xrightarrow{<I_1^+ \cup I_2^+, J_1^- \cup J_2^-, O_1 \cup O_2 \cup \{fin\}>, L_1 \cup L_2} < nothing, \emptyset >}$$

watching_stat

$$\frac{< stat, \emptyset > \xrightarrow{<I_1^+, J_1^-, O_1>, L} < stat', \eta' >}{\begin{array}{l} < \texttt{do}\ stat\ \texttt{watching}\ S\ , \emptyset > \xrightarrow{<I_1^+, J_1^-, O_1>, L} \\ \qquad < \texttt{present}\ S\ \texttt{else}\ \texttt{do}\ stat'\ \texttt{watching}\ \texttt{immediate}\ S, \eta' \uparrow S > \end{array}}$$

where $\eta' \uparrow S$ is the operation that adds signal S as a possible kill signal for the pending rendezvous labels L.

watching_immed1

$$\frac{< stat, \eta > \xrightarrow{<I^+, J^-, O>, L} < stat', \eta' >,\ S \notin I^+ \cup O}{\begin{array}{l} < \texttt{do}\ stat\ \texttt{watching}\ \texttt{immediate}\ S, \eta > \xrightarrow{<I^+, J^-, O>, L} \\ \qquad < \texttt{do}\ stat'\ \texttt{watching}\ \texttt{immediate}\ S, \eta' > \end{array}}$$

watching_immed2

$$\frac{< stat, \eta > \xrightarrow{<I^+, J^-, O>, L} < stat', \eta' >,\ E \equiv < I^+, J^-, O >, S \notin J^- \cup O}{< \texttt{do}\ stat\ \texttt{watching}\ \texttt{immediate}\ S, \eta > \xrightarrow{<E[I/I \cup \{S\}], \eta> \# S} < \texttt{nothing}, \emptyset >}$$

where $E[\cdot/\cdot]$ denotes the usual substitution;

present1

$$\frac{S \notin J_1^- \cup O_1,\ < stat_1, \emptyset > \xrightarrow{E_1 \equiv <I_1^+, J_1^-, O_1>, L_1} < stat'_1, \eta' >}{\begin{array}{l} < \texttt{present}\ S\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end}, \eta > \xrightarrow{<E_1[I_1/I_1 \cup \{S\}], \eta'> \# S} \\ \qquad < stat'_1, \eta' > \end{array}}$$

present2

$$\frac{S \notin I_2^+ \cup O_2,\ < stat_2, \eta > \xrightarrow{<I_2^+, J_2^-, O_2>, L_2} < stat'_2, \eta' >}{\begin{array}{l} < \texttt{present}\ S\ \texttt{then}\ stat_1\ \texttt{else}\ stat_2\ \texttt{end}, \eta \uparrow S > \xrightarrow{<I_2^+, J_2^- \cup \{S\}, O_2>, L_2} \\ \qquad < stat'_2, \eta' > \end{array}}$$

Table II: Inductive Rules

Signal_decl

$$\frac{< stat, \eta > \xrightarrow{E,L} < stat', \eta' >,\ S \in I^{+} \Rightarrow S \in O}{< \mathtt{signal}\ S\ \mathtt{in}\ stat, \eta \downarrow S > \xrightarrow{E/\{S\}),L} < \mathtt{signal}\ S\ \mathtt{in}\ stat', \eta' \downarrow S >}$$

where $\eta \downarrow S$ corresponds to $\{< H, \alpha \backslash \{S\} > \mid\ < H, \alpha > \in\ \eta\}$

Net_Par_Syn

$$\frac{\begin{array}{l} < P_1, \eta_1 > \xrightarrow{E_1 \equiv < I_1^{+}, J_1^{-}, O_1 >, L_1} < P_1', \eta_1' > \\ < P_2, \eta_2 > \xrightarrow{E_2 \equiv < I_2^{+}, J_2^{-}, O_2 >, L_2} < P_2', \eta_2' >,\ L_1 \cap CC = L_2 \cap CC \end{array}}{< P_1 || P_2, \eta_1 \cup \eta_2 > \xrightarrow{< I_1^{+} \cup I_2^{+}, J_1^{-} \cup J_2^{-}, O_1 - \{fin_1\} \cup O_2 - \{fin_2\} \cup fin' >, L_1 \cup L_2} < P_1' || P_2', \eta_1' \cup \eta_2' >}$$

where fin' is equal to $\{fin\}$ if $fin_1 \in O_1 \wedge fin_2 \in O_2$; otherwise, $fin' = \emptyset$.

Net_Par_Int

$$\frac{\begin{array}{l} < P_1, \eta_1 > \xrightarrow{E_1 \equiv < I_1^{+}, J_1^{-}, O_1 >, L_1} < P_1', \eta_1' > \\ \eta|_{P_1} = \eta_1,\ \eta|_{P2} = \eta'|_{P2}, \eta_1' = \eta'|_{P_1},\ L_1 \cap CC = \emptyset \end{array}}{< P_1 || P_2, \eta > \xrightarrow{E_1, L_1} < P_1' || P_2, \eta' >}$$

Hiding_initiate_1

$$< \mathtt{process\ begin}\ P\ \mathtt{end}, \emptyset > \rightarrow\ < P, \{< P, \emptyset >\} >$$

Hiding_ext1

$$\frac{< P, \eta > \xrightarrow{E \equiv < I^{+}, J^{-}, O >, L} < P', \eta' >,\ fin \notin O, kP \notin O}{< [P], \eta > \xrightarrow{< ext(I^{+}), ext(J^{-}), ext(O) >, ext(L)} < [P'], \eta' >}$$

Here ext denotes the projection of the signals to those that can be observed outside P; note that, it can share Esterel *signals. Note that there could be some some actions in L that are not observable.*

Hiding_ext2

$$\frac{< P, \eta > \xrightarrow{E \equiv < I^{+}, J^{-}, O >, L} < P', \eta' >,\ fin \in O, kP \notin E}{< [P], \eta > \xrightarrow{< ext(I^{+}), ext(J^{-}), ext(O) - \{fin\} >, ext(L)} < end_P, \eta' >}$$

Hiding_ext_ter_3

$$\frac{< P, \eta > \xrightarrow{E \equiv < I^{+}, J^{-}, O >, L} < P', \eta' >,\ kP \in E}{< [P_1], \eta_1 > \xrightarrow{E_1 \# k P_1, \emptyset} < end_P, \eta' >}$$

terminate_1

$$\frac{< P, kP > \in \eta}{< end_P, \eta > \xrightarrow{E,L} < nothing, \emptyset >}$$

Table III: Inductive Rules

Note:

1. The advantage of using end_P is that, the semantics can be easily modified for the case when the termination can be observed in the `tick` of the immediately enclosing module.
2. The semantics is consistent with the CRP semantics, in the sense, that termination of the embedded task also provides an input to the immediately enclosing module. This clearly models the asynchronous nature of the model.

3 Rules for Traps

In this section, we describe the behavioral rules of the `trap` and `exit` statements. First let us consider the priority definition among trap labels. Note that several traps may be exited simultaneously. The rule is simple: only the outermost trap matters, the other ones being discarded. For example in,

```
trap T1 in
  trap T2 in
          exit T1
         ||
          exit T2
  end ; (* trap T1 *)
  emit O
end ; (* trap T2 *)
```

the traps T1 and T2 are exited simultaneously, the internal trap is discarded and O is not emitted.

Now, let us enrich the labels of the transitions by sets of trap labels. First, we use a trap-environment which consists of sets of sequences of trap-labels encountered. Thus, the transition $\gamma \vdash < stat, \eta > \xrightarrow[T]{E,L} < stat', \eta' >$ denotes the transitions with the trap set T assuming the trap-environment γ; T is a set of trap labels that contains the labels of the `exit` statements executes by *stat*.

The rules for the trap-statement, and the exit statement are given below. For giving the flavor of the tuple corresponding to the trap set, we also describe the rule for ESTEREL parallel operator. We use the following notation in the following:

- Let γ denote an element in the set of sequence of trap labels, Γ.
- Let $initiated(I^+)$ = set of signals of the form $s\ell$ where ℓ is either an asynchronous task label or a rendezvous label. That is, $initiated(I^+)$ yields the tasks initiated in the current instant.
- The function $pending : environment \rightarrow task\_labels$ yields the set of pending tasks in the environment. That is, $pending(\eta)$ collects all the pending tasks in η.
- We define an operator *nesting* taking two trap labels and outputs the label which nests the other label. The function *nesting* is realized by operations on the trap-environment γ.

trap1

$$\frac{\gamma \circ T \ \vdash < stat, \eta > \xrightarrow[T]{<I^+,J^-,O>,L} < stat', \eta' >,}{\gamma \vdash < \mathtt{trap}\ T\ \mathtt{in}\ stat, \eta > \xrightarrow[T]{<I^+,J^-,O_1 \cup \{fin\}>\#initiated(I^+),L} < \ nothing, \emptyset >}$$

where $O_1 \ = \ O_1 \ \cup \ \{k\ell |\ \ell \in pending(\eta)\}$; *i.e., kill* signals are added for the pending tasks and the rendezvous labels under the scope of trap T. The operator $\#initiated(I^+)$ removes all the tasks initiated in the current instant.

trap2

$$\frac{\gamma \circ T \ \vdash < stat, \eta > \xrightarrow[\ell]{<I^+,J^-,O\}>,L} < stat', \eta' >, \ \ell \neq T}{\gamma \vdash < \mathtt{trap}\ T\ \mathtt{in}\ stat, \eta > \ \xrightarrow[\ell]{<I^+,J^-,O>,L} < \mathtt{trap}\ T\ \mathtt{in}\ stat', \eta' >}$$

exit

$$< \mathtt{exit\ T},\ \emptyset > \xrightarrow[T]{<\emptyset,\emptyset,\emptyset>,\emptyset} < halt, \emptyset >$$

est_par

$$\frac{\begin{array}{l} \gamma \vdash < stat_1, \eta_1 > \xrightarrow[\ell_1]{<I_1^+,J_1^-,O_1\}>,L_1} < stat_1', \eta_1' >, \\ \gamma \vdash < stat_2, \eta_2 > \xrightarrow[\ell_2]{<I_2^+,J_2^-,O_2\}>,L_2} < stat_2', \eta_2' >, \\ I_1^+ \cap J_2^- = I_2^+ \cap J_1^- = \emptyset \\ O_1 \cap J_2^- = O_2 \cap J_1^- = \emptyset \end{array}}{\gamma \vdash < stat_1 || stat_2, \eta_1 \cup \eta_2 > \ \xrightarrow[nesting(\ell_1,\ell_2)]{<I_1^+ \cup I_2^+, J_1^- \cup J_2^-, O_1 \cup O_2>, L_1 \cup L_2} < \ stat_1' || stat_2', \eta_1' \cup \eta_2' >}$$

Table IV: Inductive Rules

In rule (*trap1*), the trap exited is labelled T (below the arrow). The trap essentially preempts all the pending asynchronous tasks and the embedded statement. The killing signals corresponding to the killed tasks are added to the output signals (which is achieved through the # operator). Note that the tasks completed can in fact does so; it is here, one can distinguish *weak preemption* (i.e., trap-construct) and strong preemption (i.e., do-watching construct). In (*trap2*), η' from *stat* is carried further; the addition of kill signals will be reflected when the actual trap is encountered (that is reflected in rule *trap1*). The halt-statement rewriting into a halt is reflected in rule (*exit*); note that in this rule the *trap* emitted is T. All other rules simply propagate the trap label.

In the full paper, we discuss abstract properties of the semantic equations such as compositionality, relationship with the CSP and `meije` semantics.

3.1 Illustrative Example

We illustrate the rules given earlier by describing the operational behaviour of a simple program containing two nested watching statements:

```
do
```

```
    do
       rendezvous H
    watching S1
  watching S2.
```

Informally, the execution of this program consists of at least two steps: in the first step the rendezvous is initiated which in a latter step is either completed or preempted. The preemption can happen because of the occurrence of S1 or S2 with the latter having the priority over the former. Note that occurrence of S1 or S2 is ignored in the first step of the execution.

First Step: The behavior of the program in the first step is given by the last rewriting of the sequence of rewritings given below. This sequence describes the derivation of the last rewriting by applying the operational semantics. This derivation involves the application of '*rend_st*' rule once and '*watching_st*' rule twice.

$$\cfrac{\cfrac{< rendezvous H, \emptyset > \xrightarrow{<\emptyset,\emptyset,\{sH\}>,\emptyset} < rend\_wait H, \{< H, \emptyset >\} >}{\begin{array}{l}< \texttt{do rendezvous } H \texttt{ watching } S1, \emptyset > \xrightarrow{<\emptyset,\emptyset,\{sH\}>,\emptyset} \\ \quad < \texttt{present } S1 \texttt{ else do rend\_wait } H \texttt{ watching immediate} S1, < H, \{S1\} >\end{array}}}{\begin{array}{l}< \texttt{do do rendezvous } H \texttt{ watching } S1 \texttt{ watching } S2, \emptyset > \xrightarrow{<\emptyset,\emptyset,\{sH\}>,\emptyset} \\ \quad < \texttt{present } S2 \texttt{ else do present } S1 \texttt{ else do rend\_wait } H \\ \quad \texttt{watching immediate } S1 \texttt{ watching immediate } S2, < H, \{S1, S2\} >\end{array}}$$

Next step: The behavior of the program in the next step depends upon whether S1 or S2 occurs or the rendezvous is completed. Four cases are distinguished and in each case, a derivation sequence leading to a possible behaviour is given.

Case 1: Neither S1 nor S2 occurs and the rendezvous does not get completed

$$\frac{< \texttt{rend\_wait}\ H, < H, \emptyset >> \xrightarrow{<\emptyset,\emptyset,\emptyset>,\emptyset} < \texttt{rend\_wait}\ H, \{< H, \emptyset >\} >}{< \texttt{do rend\_wait}\ H\ \texttt{watching immediate}\ S1, < H, \emptyset >> \xrightarrow{<\emptyset,\emptyset,\emptyset>,\emptyset} < \texttt{do rend\_wait}\ H\ \texttt{watching immediate}\ S1, < H, \emptyset >>}$$

$$\frac{}{< \texttt{present}\ S1\ \texttt{else do rend\_wait}\ H\ \texttt{watching immediate}\ S1, < H, \{S1\} >> \xrightarrow{<\emptyset,\emptyset,\emptyset>,\emptyset} < \texttt{do rendezvous\_}\mathit{wait}\ H\ \texttt{watching immediate}\ S1, < H, \emptyset >>}$$

$$\frac{}{< \texttt{do present}\ S1\ \texttt{else do rend\_wait}\ H\ \texttt{watching immediate}\ S1\ \texttt{watching immediate}\ S2, < H, \{S1\} >> \xrightarrow{<\emptyset,\emptyset,\emptyset>,\emptyset} < \texttt{do do rend\_wait}\ H\ \texttt{watching immediate}\ S1\ \texttt{watching immediate} S2, < H, \emptyset >>}$$

$$\frac{}{< \texttt{present}\ S2\ \texttt{else do present}\ S1\ \texttt{else do rend\_wait}\ H\ \texttt{watching immediate}\ S1\ \texttt{watching immediate}\ S2,\ < H, \{S1, S2\} >> \xrightarrow{<\emptyset,\emptyset,\emptyset>,\emptyset} < \texttt{do do rend\_wait}\ H\ \texttt{watching immediate}\ S1\ \texttt{watching immediate}\ S2, < H, \emptyset >>}$$

Case 2: Neither S1 nor S2 occur but the rendezvous is completed.

$$\frac{< \texttt{rend\_wait}\ H, < H, \emptyset >> \xrightarrow{<\emptyset,\emptyset,\{fin\}>,\{H\}} < \texttt{nothing}, \emptyset >>}{< \texttt{do rend\_wait}\ H\ \texttt{watching immediate}\ S1, < H, \emptyset >> \xrightarrow{<\emptyset,\emptyset,\{fin\}>,\{H\}} < \texttt{nothing}, \emptyset >}$$

$$\frac{}{< \texttt{present}\ S1\ \texttt{else do rend\_wait}\ H\ \texttt{watching immediate}\ S1, < H, \{S1\} >> \xrightarrow{<\emptyset,\emptyset,\{fin\}>,\{H\}} < \texttt{nothing}, \emptyset >}$$

$$\frac{}{< \texttt{do present}\ S1\ \texttt{else}\ \ \texttt{rend\_wait}\ H\ \texttt{watching immediate}\ S1\ \texttt{watching immediate}\ S2, < H, \{S1\} >> \xrightarrow{<\emptyset,\emptyset,\{fin\}>,\{H\}} < \texttt{nothing}, \emptyset >}$$

$$\frac{}{< \texttt{present}\ S2\ \texttt{else do present}\ S1\ \texttt{else do rend\_wait}\ H\ \texttt{watching immediate}\ S1\ \texttt{watching immediate}\ S2, < H, \{S1, S2\} >> \xrightarrow{<\emptyset,\emptyset,\{fin\}>,\{H\}} < \texttt{nothing}, \emptyset >}$$

Case 3: S1 occurs but S2 does not.

$$\frac{\dfrac{\dfrac{< \texttt{nothing}, \emptyset > \xrightarrow{<\emptyset,\emptyset,\{fin\}>,\emptyset} < \texttt{nothing}, \emptyset >}{< \texttt{present}\ S1\ \texttt{else do rend\_wait}\ H\ \texttt{watching immediate}\ S1, < H, \{S1\} >> \xrightarrow{<\emptyset,\emptyset,\{fin\}>,\emptyset} < \texttt{nothing}, \emptyset >}}{< \texttt{do present}\ S1\ \texttt{else do rend\_wait}\ H\ \texttt{watching immediate}\ S1\ \texttt{watching immediate}\ S2, < H, \{S1\} >> \xrightarrow{<\emptyset,\emptyset,\{fin\}>,\emptyset} < \texttt{nothing}, \emptyset >}}{< \texttt{present}\ S2\ \texttt{else do present}\ S1\ \texttt{else do rend\_wait}\ H\ \texttt{watching immediate}\ S1\ \texttt{watching immediate}\ S2, < H, \{S1, S2\} >> \xrightarrow{<\emptyset,\emptyset,\{fin\}>,\emptyset} < \texttt{nothing}, \emptyset >}$$

Case 4: S2 occurs.

In this case, the derivation is similar to Case 3 but shorter having only the first and the last rewriting of the above sequence. Note that it does not matter whether $S1$ occurs or not.

Subsequent Steps: There would be subsequent computation steps provided Case 1 holds in the second step. Assuming this, we have the following rewritten program: '`do do rend_wait` H `watching immediate` S1 `watching immediate` S2' with the environment part being $\{H, \emptyset\}$. In subsequent steps this program is rewritten into itself with the environment remaining the same until either S1 or S2 occurs or the rendezvous is completed. When one of these events happen, the whole program terminates.

Remarks: In the above, it is assumed that S1 and S2 are distinct. If both are identical, then all the cases described above except the case that S1 occurs but S2 does not hold.

4 Verification

One of the distinct advantages of using synchronous languages for specifying reactive systems is that the description of the system analyzed/validated is very close to implementation. One of oldest languages in the family of synchronous languages Esterel has good developmental facilities such as efficient code generating compilers, verifiers etc. One of our motivation in the formulation of CRP [4] has been that we should preserve such an advantage as far as possible. In this section, we illustrate that it is indeed the case that the same verifying techniques/tools used on Esterel programs are also usable on CRP programs.

In the following, we describe briefly the compositional verification of Esterel before we show how the same techniques can be applied to hierarchical CRP and illustrate the verification on the banker-teller example using the system Auto/Autograph (or mauto/atg) described in [12].

4.1 Compositional Verification of Esterel

In [11], a compositional semantics of Esterel has been defined which enables compositional verification. The structure of the transitions has been discussed in the previous section that gives the power of compositionality for ESTEREL programs. The main idea behind the verification is that ESTEREL programs can be compiled into reactive automata ([11]) which can then be compared with specification automata. The power of "compositionality" for verification lies in applying transformations [11] such as:

1. Behaviour Abstraction: One of the most useful transformations of this class is *signal hiding*. In fact, we can define this transformation with reference to the set of signals say S on each of the syntactic constructs of the language. Essentially, the transformation removes the signals not in S from the transition events. Thus, the resultant automata may become nondeterministic keeping the number of states the same (hence, reachability properties would be preserved).

2. Symbolic Bisimulation: It is possible to define a bisimulation which will be congruent to the ESTEREL operators and commute with the hiding transformation.

3. Filtering: Better reductions can be achieved by guessing local contexts.

The above method will not be usable unless it is scalable. It is here the aid of tools comes to rescue. The verification system called Auto (the current version of this called Mauto) can be effectively used in checking the above transformations. A variety of automata operations with reference to the above transformations can be used to manipulate and compare automata. Some of the interesting operations are: bisimulation equivalence checking, deadlock detection, hiding and abstraction. An associated system Autograph/Atg ([12]) provides a user-friendly visual aid for viewing these automata. Using this system, Esterel programs can be verified to satisfy desired properties. In the context of verification, one major problem is the state-space explosion. A number of proposals are being offered in the literature. The above technique overcomes this problem as the transformations mentioned above lead to simplifications of the automata being verified.

4.2 Verifying CRP programs

The main issue lies in establishing that the compositional reductions can be achieved at the ESTEREL node level and the network level. The crux of verification lies in synthesizing *reactive automata* from the semantics. In the context of CRP, a transition depends on an event composed of (1) necessarily present signals, (2) necessarily absent signals, (3) output signals, and (4) the set of active/pending rendezvous actions which could be preempted. Since variables are not shared and the channels connect at most two processes, we can

1. Synthesize a local reactive automata at each node.

2. Synthesize a network reactive automata that reacts with reference to the rendezvous actions which can be locally preempted.

Let us consider the reactive automata at a node. It is obvious that each is locally reactive and the `ticks` are distinct at each node; they can synchronize only through the rendezvous actions. Thus, the reactive automata does not really depend upon the number of nodes but only on the number of nodes with which it interacts. The semantics outlined above enables to use the transformations described above both at the node and the network level.

As mentioned already, for a given CRP program there is an associated *reactive automata* which can be constructed using the operational semantics of CRP; the nodes in this automata corresponds to the CRP programs and its derivatives while the transitions capture the 'arrow' relation defined by the semantics. The transitions are more complex than the ESTEREL transitions as there is an additional component of completed rendezvous actions. This extra structure does not cause much problem in the verification process using Auto thanks to the underlying powerful meije calculus [10]. In fact, it contributes to a structuring of the proof process itself as follows:

- A CRP program can be abstracted out to be an ESTEREL program by hiding all the rendezvous actions and signals.
- A CRP program can be abstracted out to be a distributed program by hiding all the ESTEREL signals and retaining only the rendezvous actions.
- A CRP program can be viewed as a program consisting of processes interacting via either ESTEREL signals or rendezvous actions.

Depending upon the property under verification, appropriate abstractions can be carried out. This flexibility is possible because the semantics of CRP interprets rendezvous actions in terms of Esterel signals. As mentioned above, we can effectively use the above mentioned transformations for verifying CRP programs. An interesting and important observation of this paper is that these transformation rules are applicable to CRP programs as well and hence, they can help in containing the state explosion problem.

4.3 Illustrative Example of Verification: A Banker-Teller Program

As an illustration of the verification process, we shall take the Banker Teller example discussed in the original paper [4] and verify its correctness. This program consists of two CRP processes, Teller and Bank. Teller controls a machine through which the clients of the bank interact and carry out transactions. Teller processes controlling different machines interact with the Bank node which actually performs the money transfers. Each of these nodes, in addition to being locally reactive, interact with other nodes through rendezvous actions.

The teller machine reads a card, checks the code locally, and communicate by rendezvous with the bank to perform the actual transaction. During the whole process, the user can cancel the transaction by pressing a `Cancel` key.

The reactive interface is self-explanatory. The rendezvous interface consists of three valued channels. The output channels `CardToBank` and `AmountToBank` are used to send the card information and transaction amount. The input channel `Authorization` is used to receive a boolean telling whether the transaction is authorized. No rendezvous label is necessary here since there is exactly one `rendezvous` statement per channel. The output channel `CancelToBank` is used by the machine to inform Bank that the user has cancelled the current transaction.

The body repeatedly waits for a card and performs the transaction. The client-code checking section is purely local and reactive. The body after reading the valid code waits locally for the amount to be given (typed) by the client. Subsequently, the information about the card and amount are sent through the respective channels to Bank. Then the machine waits for authorization signal from the bank. This is a boolean valued signal and if it is true, one instantaneously delivers the money and returns the card exactly when the authorization rendezvous is completed, to ensure transaction atomicity. During the transaction, the user can press the `Cancel` key, in which case the transaction is gracefully abandoned, including the various rendezvous, and the card is returned. Since the bank considers the transaction to be performed when the `Authorization` rendezvous is completed, one must declare an exclusion relation between `Authorization` and `Cancel` to prevent these two events to happen simultaneously. The outer trap handles the abnormal cases of the transaction: the card is kept if the code is typed in incorrectly three times or if the transaction is not authorized. We use a full ESTEREL extension of the `trap` statement, in which one can write a trap-handler to be activated when the trap is exited. This extension is easily derivable from kernel statements.

In the code given below, we do not use explicit channel-declarations and further, we use the syntax of the prototype CRP compiler for the rendezvous statements. That is,

```
exec RdvSendCardToBank () (?Card) return CardToBank
```

stands for

```
rendezvous_send CardToBank (?Card) return  CardToBank
```

The statement denotes sending the value in the signal, Card, over the channel, *CardToBank*, and returns the pure signal CardToBank; note that we have not made distinctions between the channel name and the type of return signals. Similarly,

```
exec RdvRecvAuthorization()() return Authorization
```

stands for

```
exec RdvRecvAuthorization()() return Authorization
```

Here, the statement denotes receiving a value on the channel, Authorization, and the return signal is a pure signal Authorization. The other statements are similarly interpreted.

```
module BankerTeller :
type CardInfo; function CheckCode (CardInfo, integer) : boolean;
input Card : CardInfo;
input Code : integer, Amount : integer;
input Cancel;
output GiveCode, EnterAmount;
output DeliverMoney : integer;
output KeepCard, ReturnCard;
task RdvSendCardToBank () (CardInfo);
task RdvSendAmountToBank () (integer);
task RdvRecvAuthorization () () : boolean;
task RdvSendCancelToBank () ();
return CardToBank, AmountToBank;
return Authorization : boolean;
```

```
return CancelToBank;
relation Cancel # Authorization;
   loop
      trap ReturnCard, KeepCard in
         await Card;
                  do
            trap CodeOk in
                  emit GiveCode;
                  await Code;
                  if CheckCode(?Card, ?Code) then exit CodeOk end
            exit KeepCard
            end trap;
         watching Cancel timeout
            exit ReturnCard
         end;
         do (* watching Cancel again, but this time provokes *)
            (* cancellation rendezvous with the bank*)
            [
               exec RdvSendCardToBank()(?Card) return CardToBank
            ||
               emit EnterAmount;
               await Amount
            ];

            exec RdvSendAmountToBank()(?Amount) return AmountToBank;

            exec RdvRecvAuthorization()() return Authorization;
            if ?Authorization then
               emit DeliverMoney(?Amount);
               exit ReturnCard
            else
               exit KeepCard
            end if
         watching Cancel timeout
            exec RdvSendCancelToBank () () return CancelToBank;
            exit ReturnCard
         end;
      handle ReturnCard do
         emit ReturnCard
      handle KeepCard do
         emit KeepCard
      end trap
   end loop
end module
```

The part of code of Bank that interacts with one Teller machine is given by the following segment:

```
module Bank:
type CardInfo;
output CARD_INFO: CardInfo;
output AMOUNT: integer;
```

```
input ACCEPT, REFUSE;
task RdvRecvCardToBank () () : CardInfo;
task RdvRecvAmountToBank () () : integer;
task RdvSendAuthorization () (boolean);
task RdvRecvCancelToBank () ();
return CardToBank: CardInfo;
return AmountToBank: integer;
return Authorization;
return CancelToBank;
loop
   [ do
         exec RdvRecvCardToBank()() return CardToBank;
         emit CARD_INFO(?CardToBank);
         exec RdvRecvAmountToBank()() return AmountToBank do
         emit AMOUNT(?AmountToBank);
         var Acceptation : boolean in
            await
                case ACCEPT do Acceptation := true
                case REFUSE do Acceptation := false
            end await
            exec RdvSendAuthorization()(Acceptation) return Authorization
         end var
     watching CancelToBannk
   ||
     do
        exec RdvRecvCancelToBank()() return CancelToBank do
     watching Authorization
   ]
end loop
end module
```

Using the operational semantics, the automata for the two processes can be obtained. The bank node can interact with a number of teller nodes (a fixed number) but for the purpose of verification, it is sufficient to look at the combined behavior of Bank with one Teller process. This is achieved by subjecting the Bank node to the behavioral abstraction transformation that hides all the signals of Bank other than those that are required for the interaction of one Teller process. After this abstraction, the Bank automata and the Teller automata are constructed. Now we can construct the automata describing the behavior of the Bank and the Teller; the automaton is shown in Figure 1 (for the sake of brevity, we have only shown the combined (merged) automata). This is rather large and can be abstracted out to a simple automata shown in Figure 2. The transitions in this automata are obtained by renaming and combining the transitions of the original automata. The exact relationship between the actions of the original automaton and that of the new one is given in the syntax of Auto as follows:

```
parse-criterion CR=
CARD = (/Card? and not /Cancel?),
RT-CODE = (/Code? and not /Cancel? and not /KeepCard!),
AMOUNT = (not /Cancel?):(/Amount? and not /Cancel?),
CANCEL = (/Cancel?):(not /CancelToBank?)*:(/CancelToBank?),
```

```
ACC = (not /Cancel?):(/ACCEPT? and not /Cancel?),
REF =(not /Cancel?):(/REFUSE? and not /Cancel?),
WRONG-CODE = (/Code? and /KeepCard!),
NO-MONEY = (/KeepCard! and /Authorization?),
SUCCESS = (/Authorization? and /DeliverMoney!);
```

The renamings given in the parse-criterion follow the usual regular expression syntax of Auto. For the sake of completeness, we briefly explain the parse-criterion:

The parse criterion is named CR and is a sequence of equations that specifies that any path in the original automata satisfying the right hand side of an equation should be replaced by an edge labeled by the corresponding left hand side in the new automaton. The right hand side of an equation is a regular expression over boolean expressions with ':' being the concatenation operator and '*' being the Kleene-closure operator. Any boolean expression on the right hand side is made up of basic propositions of the form $\backslash a$ where, a, is a signal. An edge satisfies the proposition $\backslash a$ provided the meije action labeling the edge contains signal a.

For example, the following equation

AMOUNT = (not /Cancel?):(/Amount? and not /Cancel?),

specifies that the new automaton has an edge labeled 'AMOUNT' in the place of a path in the old automaton, that consists of two edges, the label of the first edge not containing signal $Cancel$ and the second edge containing $Amount$ but not containing $Cancel$.

It is straightforward to see that the abstracted automaton with the parse criterion, CR, given above has the intended behavior of a correct Banker-Teller.

5 Discussion

In the previous sections, we have described a compositional behavioural semantics of CRP with nested network parallelism. The semantics given here differs from the semantics given in [4] is two aspects: (i) we no longer use "decoration of labels" and (ii) the transition labels consist of both *present* and *absent* information. We also do not use the termination levels as it leads to problems in compositional semantics; note that, the termination levels provide a convenient means of encoding traps. The nested parallelism permits hierarchical abstraction. We have also illustrated how the techniques of verification of ESTEREL programs can be adapted for CRP (with the hierarchy operator) programs. Further, through the example of banker-teller described in [4], we have shown how Auto/Autograph can be used for the verification of CRP programs. Since the verification is done at the reactive automata level we can also use it for timed CRP which enables specification of hybrid systems.

It may be noted that having given the semantics of hierarchical parallel operator, it is easy to provide a semantics for creation of objects. In other words, we can obtain a compositional semantics of object-oriented CRP. One of the reasons why we have not given the semantics of objects here is that it is not clear as to how to verify the resultant object CRP programs. It may be pointed out that we could effectively use the techniques of ESTEREL verification for the hierarchical CRP programs, as we could map the system into a set of interacting reactive automata expressible in terms of the powerful *meije* calculus [10] on which the verification tools have been built. However, with the additional notion of objects, we cannot use *meije* calculus

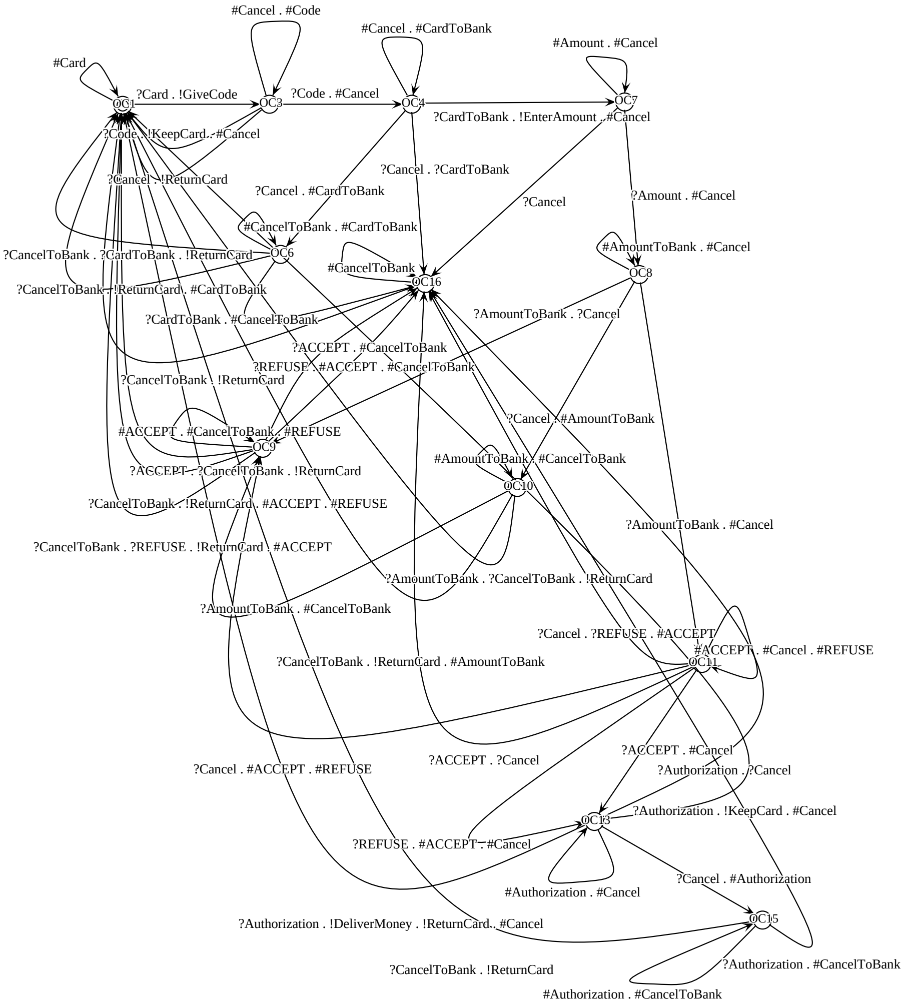

Fig. 1. Combined Automata

as the underlying calculus and hence, we have not been able to effectively use the verification techniques. We are also working[5] towards an abstraction of ESTEREL with multiple clocks with the idea that overcoming some of the drawbacks of the current module definitions of ESTEREL and also to provide a integrated formalism for perfect synchrony, asynchrony and objects.

Acknowledgments

It is great pleasure to thank Professors G. Berry, F. Boussinot and R. de Simone of CMA, Sophia Antipolis for invaluable discussions.

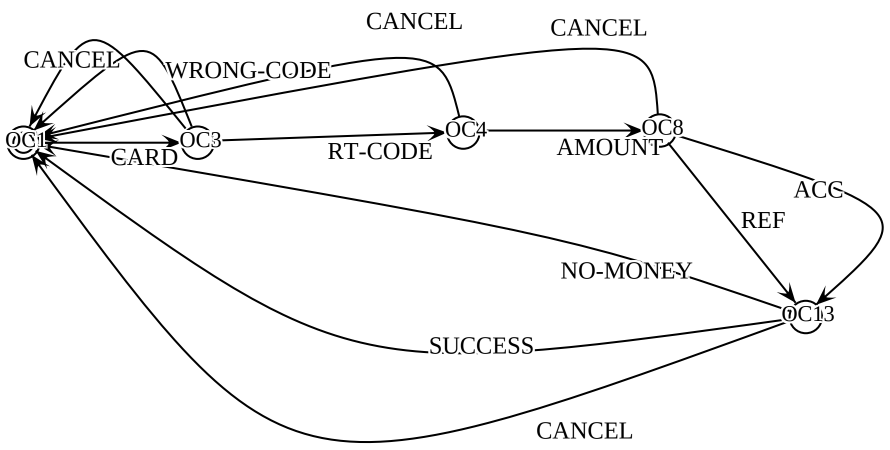

Fig. 2. Abstracted Automata

References

1. G. Berry (1992), *A hardware implementation of pure Esterel*, Sadhana: Academy Proceedings in Engineering Sciences, Indian Academy of Sciences, Special Issue on Real Time Systems, edited by RK Shyamasundar, 17 (1):95-139, 1992.
2. G. Berry and G. Gonthier, *The Esterel synchronous programming language: Design, semantics, Implementation*, SCP, Vol. 19, No.2, Nov. 92, pp. 87-152.
3. G. Berry, *Causality in* ESTEREL *Programs*, Dagstuhl Seminar on Synchronous Languages, 28 Nov - Dec 2'94.
4. G. Berry, S. Ramesh and R.K. Shyamasundar, *Communicating Reactive Processes*, 20th ACM POPL, South Carolina, Jan. 1993, pp. 85-99.
5. G. Berry, S. Ramesh and R.K. Shyamasundar, *Multi-Clock ESTEREL*, manuscript, December 1994.
6. O. Ploton, *Identifying Causality in* ESTEREL *Programs*, Dagstuhl Seminar on Synchronous Languages, 28 Nov - Dec 2' 94.
7. R.K. Shyamasundar, *Specification of Hybrid Systems in CRP*, Proc. of AMAST 93, Workshops in Computing Series from Springer-Verlag, Edited by M. Nivat, C. Rattray, T. Rus and G. Scollo, pp. 227-238, December 1993.
8. R.K. Shyamasundar, *Specification of Dynamic Real-Time Systems in CRP*, Proc. IFIP 94, Hamburg, Aug. 28 - Sept 2'94.
9. R.K. Shyamasundar and S. Ramesh, *Languages for Recative Specifications: Synchrony vs Asynchrony*, Proc. FTRTFT, Lubeck, LNCS, 863, Sept 1994.
10. R. de Simone, *Higher Level Synchronizing devices in Meije-SCCS*, TCS, 37, pp 245-267, 1985.
11. R. de Simone and A. Ressouche, *Compositional Semantics of ESTEREL and Verification by Compositional Reductions*, Proc. of Computer Aided Verification 94, June 94.
12. V. Roy and R. de Simone, *Auto and Autograph*, In R. Kurshan, editor, *proceedings of Workshop on Computer Aided Verification*, New-Brunswick, June 1990.