COVERS - A Tool for the Design of Real-Time Concurrent Systems

A.V. Borshchev[1], Yu.G. Karpov and V.V. Roudakov
Technical Cybernetics Department
St.Petersburg State Technical University
195251 St.Petersburg Russia
E-mail: covers@dcs.stu.spb.su

We give an overview of the existing commercial tools for the design of real-time concurrent systems and propose a list of features the ideal design environment should have. Then the COVERS tool is described. Its underlying abstract model is a derivative of the Timed Transition System. The concrete modelling language is based upon the structural, behavioral and data processing views on the real-time concurrent system. COVERS supports a sequential subset of Statecharts and ANSI C. The formal semantics of the concrete model is presented. We describe a method of testing the real time temporal properties using spy processes, and, briefly, debugging and performance analysis means.

1 Introduction and overview of existing tools

A lot of papers has been written on the importance of using the executable models and the corresponding tools in the design of real-time concurrent systems. However, so far none of the existing tools is being seriously used as a real design instrument. The reasons should be partly clear from a short overview we give below.

The most popular tool is, probably, the **BONeS Designer** [BONeS]. A system under development is described there as a hierarchical data flow diagram. A block is triggered as the data items arrive to its input ports. There are hundreds of standard blocks in the BONeS library, however, an experienced user can try to write an algorithm of the primitive block in C. The types of data items are organized in a hierarchical structure with a sort of limited inheritance. The other basic entities include resources (server-type and quantity-shared), memories and events, each having some standard blocks to work with it. The simulation algorithm is based on the event calendar containing asynchronous events which are scheduled by such blocks as delays. An asynchronous event may cause several instantaneous synchronous events. The probing mechanism is provided to collect statistics.

The main disadvantage of the tool is that the user is forced to express everything in terms of the data flow diagrams. When it comes to the

1. Until March, 1996 with Hewlett-Packard Laboratories, Bristol. E-mail: avb@hplb.hpl.hp.com

sequential algorithms or data processing, the model tends to look huge and very unnatural. Further, BONeS tries to provide its own special language for every king of thing, like data types or expressions. This increases the distance between the model and the real object. For instance, the data types in the main hierarchy do not correspond to the data types of C (that is used in other parts of the tool), or any other programming language. The semantics of block diagram also causes inconveniences for the user, who has to care of data losses, "stale" data, racing, etc. caused not by the nature of the system he models. Recently (in version 3.0) the finite state machines were introduced for the block behavior description, however, the way they are implemented leaves much to be desired. Unfortunately, all this moves BONeS towards an eclectic combination of discordant approaches and ideas.

The model in **SES Workbench** [SES] consists of one or several submodels, each represented by a directed graph. The basic elements are *nodes*, *arcs*, *transactions* and *resources*. Nodes correspond to a certain stage in the transaction's life: service, delay, allocation or release of resource. Transactions (representing messages, jobs, processes, etc.) travel from node to node along the arcs. Having arrived to a node, transaction is either serviced or, if the node is busy, joins the queue. An arbitrary data structure can be associated with transaction. Among the 24 basic types of nodes in SES Workbench there are resource control nodes, transaction flow control nodes, nodes for work with submodels, and so on. The user can create his own nodes in object-oriented manner and define references to nodes. A transaction arriving at a node is considered as calling a method of a node. The statistic collection is bound to either nodes or transaction categories, for instance: the time interval between two subsequent arrivals of transaction to a particular node, amount of available resources, the time transaction waits in a queue. An assertion can be set at any place, and it will be tested each time a transaction crosses the place. The simulation algorithm is pretty much the same as the one of BONeS.

As one can see, SES Workbench uses the same dataflow diagram concept as BONeS, having, however, less flexibility. The main limitation of SES Workbench is again the absence of natural means for the description of control flow. This leads to an awkward models. For example, in [P92] the network communication object is modelled by a loop of nodes with a special transaction defining the location of control.

Statemate [HLNPPSST90] is based on more modern ideas in the reactive system design. Following the statement that the functional model should be supplemented with the behavioral one, the authors of the tool implemented the Statecharts formalism [H87] - a serious attempt to structurize the behavior description. The functional structure of the system under development is specified as a set of communicating activities with which the Statecharts can be associated. In Statecharts, the states of finite-

state machines can be substituted for by a state machine of a new level, or even a set of concurrent state machines. The first option is definitely useful when one describes the different operation modes of the system, or just a reaction on a certain event that is common for a subset of states. Also, more flexibility is added to a description technique by allowing the transitions to cross the hierarchy boundaries. Concurrent parts of the system communicate using the broadcast. The transitions of the state machines are triggered by broadcast signals or timing events. A guard expression and an action can be associated with a transition. In addition to the common simulation tool functionality, Statemate can try to construct the reachability graph of the system behavior and thus analyse the reachability properties. However, this is possible for toy models only.

Among the reasons why the Statemate is not widely used, the main ones, to our mind, are a very restricted communication mechanism having sophisticated semantics, and a poor language for data manipulation.

It is also worth to mention the tool called **Design/CPN** [DCPN]. Its underlying mathematical model is a Petri net overgrown with a huge number of extensions. The transitions can be hierarchically decomposed into new nets, the tokens are coloured, the guard expressions can be associated with transitions. The real time is introduced by assigning delays to transitions and arcs, and timestamping tokens.

Not surprisingly, none of the efficient Petri net analysis techniques can be applied to the resulting object called HCPN, and the main question arising here is: what is reason for using Petri nets then? The only thing available is the construction of the occurrence graph (the graph of reachable markings) according to the HCPN operational semantics, and it is implemented in the tool. Again, for practical systems the occurrence graph is too big. The other problem is the use of Standard ML as the data processing language, which is nice, but too academic.

Summarizing this overview we propose the list of features the ideal real-time concurrent systems development environment should have:

1. Two separate modelling languages: one for the description of the structure of communicating activities and the flow of data between them, and the other one - for the description of behavior of these activities, i.e. the flow of control inside them.

2. Possibility to specify different communication disciplines. At least shared data, synchronous communication and message passing should have easy representation.

3. Simple means to express the basic elements of reactive behavior: the state of waiting for the several alternative events, timeouts and delays.

4. Powerful standard data processing language conveniently embedded into the structure / behavior specification.
5. Hierarchy for both functional structure and behavior description techniques.
6. Possibility to define types of activities and, probably, behavior elements and refer to them.
7. Scalability. The user should be able to define a variable size periodic structure of communicating activities.
8. Rigorous formal semantics for all elements of modelling language. The semantics should clearly define the time model, duration of actions and communications, level of atomicity, simultaneity, nondeterminism.
9. Executability. The ability to obtain any of the possible trajectories of the system behavior.
10. Visualization of the system behavior and "source code" debugging facilities.
11. Requirement specification language covering real time temporal properties of the system. Possibility to verify and/or test these properties.
12. Convenient performance analysis tools.
13. The straightforward translation into the implementation language wherever possible.

The COVERS tool described in this paper is designed to meet most of these requirements.

2 Abstract model

The COVERS underlying model has two levels: abstract model and concrete model. The first one is a mathematical abstraction dealing with fundamental issues of the discrete real-time behavior. It is not directly visible to the user. The concrete model captures such aspects of the system under development as concurrency, data flow, communication, control flow, data processing etc. The user works only in terms of concrete model.

The object used as an abstract model is derived from the Timed Transition System [HMP92] and is called TTS below. It is a tuple

$\langle \Sigma, s^0, T, C, d \rangle$, where

Σ is (possibly, infinite) set of states,
$s^0 \in \Sigma$ is the initial state,
$T \subseteq \{\tau | \tau : \Sigma \to \Sigma \cup \{Null\}\}$ is a finite set of transitions,
$C \subseteq T \times T$ is a conflict relation,
$d : T \to R^{\geq 0}$ is a function assigning a *firing delay* to each transition.

Time in TTS is measured by real numbers. Transitions are defined as functions on the set of states. Transition τ is called *enabled* in the state s if $\tau(s) = s' \neq Null$. A set of all transitions enabled in the state s is denoted by $T(s)$. A transition can be *taken* (can fire) only when it is enabled. Transition firing takes zero time. When τ is taken in the state s, the next state of the system is $\tau(s)$. In between two subsequent transition firings the system state remains the same. The important property of TTS is that transitions in general are not alternative to each other. They can fire independently, and this enables us to model the concurrent systems.

It might happen that between the moment the transition τ becomes enabled and the moment it is taken, the other transitions fire. Some of them may disable τ, the others - do nothing to it, and the third - disable and re-enable it at once. To make distinction between two last cases we introduce the *conflict relation*. The way it works is explained later.

The operational meaning of the TTS is captured by the notion of *computation* - a sequence of *situations* and *steps*:

$$\langle s_0, t_0 \rangle \xrightarrow{\tau_0} \langle s_1, t_1 \rangle \xrightarrow{\tau_1} \ldots \to \langle s_i, t_i \rangle \xrightarrow{\tau_i} \ldots,$$

where $s_i \in \Sigma$, $s_0 = s^0$, $t_i \in R^{\geq 0}$, $\forall i : t_i \leq t_{i+1}$ and $\tau_i \in T \cup \{\varepsilon\}$.

The situation $\langle s_i, t_i \rangle$ means that at time t_i the system enters the state s_i. The step τ_i which is taken at t_{i+1} drives the system into the state s_{i+1}. A step can be either the transition firing ($\tau_i \in T$) or the *timed step* denoted by ε. In the following definitions ε is not in conflict with any of the transitions.

Transition $\tau \in T$ *becomes enabled* at the situation $\langle s_i, t_i \rangle$ if $\tau(s_i) \neq Null$ and either $i = 0$ or $\tau(s_{i-1}) = Null$ or $\langle \tau_{i-1}, \tau \rangle \in C$.

Transition $\tau \in T$ *is taken* at the situation $\langle s_i, t_i \rangle$ if $\tau_i = \tau$.

Transition $\tau \in T$ *is continuously enabled* from $\langle s_i, t_i \rangle$ to $\langle s_j, t_j \rangle$, $i \leq j$ if $\forall k : i \leq k \leq j : \tau(s_k) \neq Null$ and $\forall k : i \leq k < j : \langle \tau_k, \tau \rangle \notin C$.

The computation must satisfy the following three conditions.

1. $\forall i$ if $\tau_i \in T$ then $s_{i+1} = \tau_i(s_i)$ and $t_{i+1} = t_i$. Otherwise, if $\tau_i = \varepsilon$ then $s_{i+1} = s_i$ and $t_{i+1} > t_i$. Also, we require that if $\tau_i = \varepsilon$ then $\tau_{i+1} \neq \varepsilon$. This means that a computation consists of alternating *state-changing*

and *time-changing* phases. In a state-changing phase, which takes zero time, several transitions can be taken. In a time-changing phase the time progresses, and the system state remains the same.

2. If $\tau \in T$ is continuously enabled from $\langle s_i, t_i \rangle$ to $\langle s_j, t_j \rangle$, $i \le j$ then $t_i + d(\tau) \le t_j$. Thus, any transition can not be continuously enabled longer than is defined by its firing delay. If $d(\tau)$ passes since τ has been enabled, it should be either disabled or taken.

3. If $\tau \in T$ becomes enabled at $\langle s_i, t_i \rangle$, is continuously enabled from $\langle s_i, t_i \rangle$ to $\langle s_j, t_j \rangle$, $i \le j$, and taken at $\langle s_j, t_j \rangle$, then $t_i + d(\tau) = t_j$. Thus, a transition can not be taken before its firing delay expires.

This definition of computation is, however, not constructive. To be able to build any computation of the TTS we will extend the notion of situation.

A *full situation* is a triple $\langle s, t, r \rangle$, where $\langle s, t \rangle$ is a situation, i.e. a state and a time it has been entered, and $r: T(s) \to R^{\ge 0}$ is a function assigning a *remaining firing time* (or, simply, *remainder*) to each transition enabled in s.

Obviously (we also can refer the reader to the [AH92]), the full situation fully defines the future behavior of the system. The following algorithm is used in COVERS to obtain the computations of the TTS.

Let $\langle s, t, r \rangle$ be the current full situation, and τ_{next} - the next step.

// initially, remainders are set to the firing delays
$\langle s, t, r \rangle = \langle s^0, 0, r^0 \rangle$, **where** $\forall \tau \in T(s): r^0(\tau) = d(\tau)$;
while($T(s) \neq \emptyset$ **)** { *// while some transitions are enabled*
  $r_{min} = min(r(\tau) | \tau \in T(s))$; *// we get the minimum remainder*
  if($r_{min} > 0$ **)** { *// if no one is ready to fire*
    // a timed step is made
    $\tau_{next} = \varepsilon$;
    // and remainders are decreased by r_{min}
    $\langle s, t, r \rangle = \langle s, t + r_{min}, r' \rangle$, **where** $\forall \tau \in T(s): r'(\tau) = r(\tau) - r_{min}$;
  } **else** { *// if someone is ready to fire*
    τ_{next} = **any** $\tau \in T(s)$ **such that** $r(\tau) = 0$;
    // the state changes instantaneously
    $\langle s, t, r \rangle = \langle \tau_{next}(s), t, r' \rangle$, **where** $\forall \tau \in T(\tau_{next}(s))$:
      // for the transitions that become enabled
      // remainders are equal to their firing delay
      $r'(\tau) = d(\tau)$ **if** $\tau(s) = Null \vee \langle \tau_{next}, \tau \rangle \in C$, **and**
      // for the transitions that remain enabled
      // remainders are not changed
      $r'(\tau) = r(\tau)$ **otherwise;**
  }
}

The set of all sequences of full situations and steps generated by this algorithm coincides with the set of all possible computations of the TTS, or its *behavior*.

It is interesting to consider the possibility of finite representation of the system behavior. A directed graph with pairs $\langle s, r\rangle$ as vertices and steps τ as arcs could serve as such a representation. Unfortunately, even if the TTS has a finite number of states, the set of reachable pairs $\langle s, r\rangle$ can be infinite. The simplest example is the following TTS: $\Sigma = \{s\}$, $s^0 = s$, $T = \{a, b\}$, $a(s) = b(s) = s$, $C = \varnothing$, $d(a) = \sqrt{2}$ and $d(b) = \sqrt{3}$. Since $\sqrt{2}$ and $\sqrt{3}$ do not have a common multiple, the set of reachable pairs $\langle s, r\rangle$ is infinite. This frustrating result applies generally to all models capable to express this two primitive concurrent cyclic transitions. If we restrict ourselves by considering *rational* firing delays only, the behavior will always have a finite representation. However, it still can be too big to be constructed by a computer, and, moreover, by making such an approximation we can loose some important branches.

Consequently, any verification of real-time concurrent systems based on the construction of all reachable situations is practically impossible. In the paper we mainly concentrate on other analysis methods such as testing of temporal properties.

3 Concrete model

The concrete model, or the modelling language of COVERS is based upon the three views on the system under development: structure, behavior and data processing. From the point of view of functional structure the system is represented by a set of concurrent communicating *processes*. *Extended state machines* (visually being a "sequential" Statecharts [H87]) are used to specify the behavior of processes. All processing of data (sending, reception, testing and modification) is specified in *C* language.

A *process* is a sequential object driven by the external events like communication with another process or elapsing of the specified amount of time. We assume that there is a global clock in the system, and also that each process is "executed on its own processor".

Three disciplines of process communication are provided by COVERS as basic: synchronous communication (*rendezvous*), asynchronous message passing (*cast*) and *shared variables*. This set was chosen for the following reasons. First, all three types can be frequently met in real-life systems. Second, they are a sort of orthogonal, since it is hard to express one of them in terms of the others. Finally, more sophisticated communication mechanisms can be easily constructed on top of them.

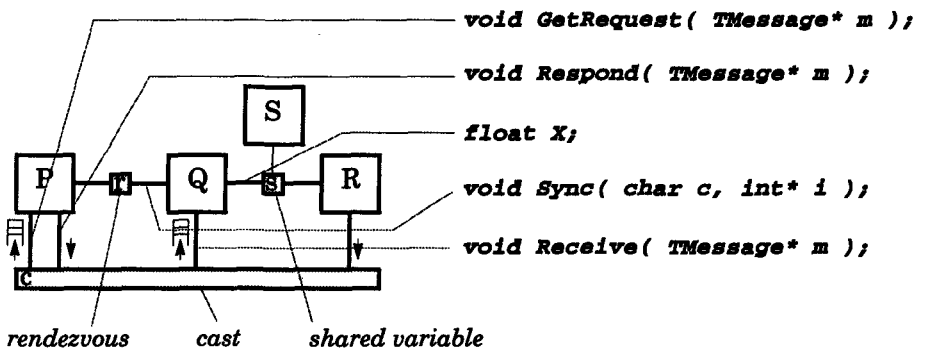


Fig. 1. Structural view on the system under development

An example of structure chart is shown in Fig. 1. Processes P, Q, R and S are connected to the communication objects with *connectors*. A C declaration associated with a connector defines how the process will access the communication object. In case of shared variable it is just a variable declaration, in case of rendezvous or cast it is a function declaration.

Synchronization of two processes, or rendezvous, is similar to the rendezvous of the Ada language. It occurs immediately as soon as both processes are ready, and takes zero time. If one process is ready, and the other is not, the first one has to wait for the partner - see Fig. 2. Bidirectional data exchange can be performed during the rendezvous. The exchange algorithm can be specified by the user in C.

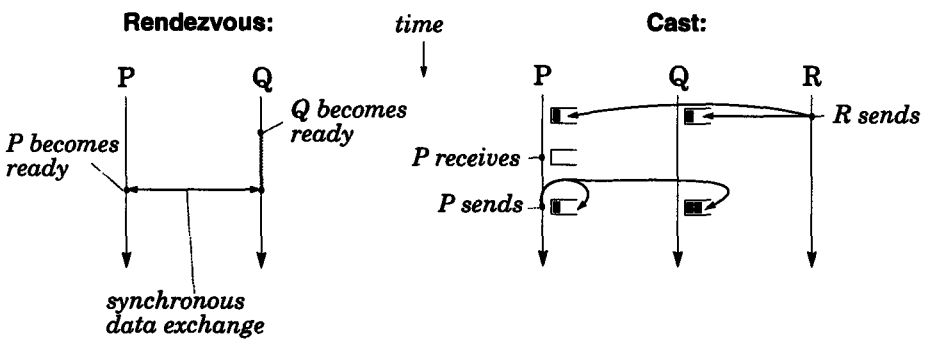


Fig. 2. Synchronous and asynchronous communication

Cast is a message passing mechanism with queuing. It is asynchronous in the sense that the sender can send a message at any time, it does not care whether the receivers are ready. Cast communication objects may connect arbitrary number of processes. A process can be connected to the cast as a sender, as a receiver, or both. Having been sent, the message immediately appears in the input queues of all receivers - see Fig. 2. Having been

received by the process, the message is removed from its queue. If the process wants to receive a message, but the queue is empty, it waits. The messages of a particular cast object are of the same C data type.

A process connected to a shared variable can access it at any time. The operation on shared variable performed during a single system step is considered as atomic. An access to a shared variable as well as an act of sending a cast message take zero time.

The behavior of a process is described in terms of *states* and *transitions*. Being in a state, the process is waiting for some external event to occur. The transitions outgoing the state define the set of events the process will react to, and the reaction itself. A transition has three attributes: *guard*, *event* and *action*. A guard is a Boolean C expression over the process' private variables and variables shared with other processes. All guards of the transitions outgoing the current state are checked "continuously". A transition can only fire when its guard evaluates to TRUE.

Three possible kinds of events can be associated with transitions. The first one is a reception of the cast message, the second - a rendezvous with other process, and the third - elapsing of the specified amount of time (time delay). Syntactically first two kinds look like a call to a corresponding function, and the last one - as an arbitrary C expression of the float type.

Action is a piece of C code describing the process' reaction on the event. It may include the modification of private and shared variables and sending cast messages.

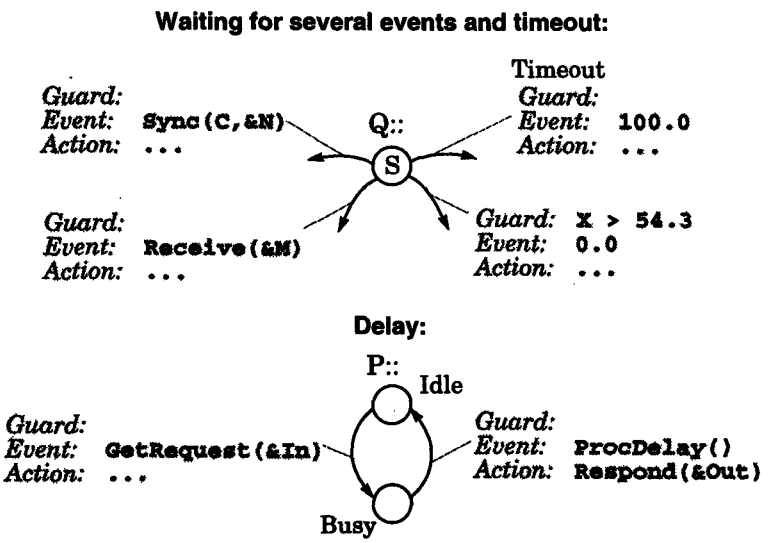


Fig. 3. Behavior. Basic elements

A transition fires instantaneously as soon as the corresponding event is available. If the time delay *t* is associated with a transition, transition will fire provided the guard is *continuously* TRUE for *t* time units.

Fig. 3. shows how the COVERS modelling language represents the basic elements of the reactive behavior. The process *Q* (its external connections are shown in Fig. 1.) in the state *S* waits for the rendezvous with process *P*, the cast message from *P* or *R*, and for the shared variable *X* to become greater than 54.3. Note that the waiting for a condition *c* to become TRUE is specified as a transition with the guard *c* and zero delay. If none of these events are available within 100.0 time units since *Q* has entered the state *S*, the *Timeout* transition is taken. The "pure delay" is shown in the behavior of the process *R* below. Having received the cast message, *R* comes to state *Busy*, spends there the time *ProcDelay()*, then sends some message back and returns to *Idle*.

Besides the plain state diagrams, COVERS supports the following Harel's extensions [H87]: hyperstates, history states and branching of transitions. Hyperstate is a compact representation of the group of states having the identical reaction on a particular event. Consider the model of a process with failures in Fig. 4. The hyperstate Working corresponds to the normal operating mode, and in the state *Crashed* the process is crashed. The transition *Crash* may be taken no matter what the process is doing, i.e. in either of the states *S1*, *S2* or *S3*. If after the recovery the process wants to return to the state where its operation has been interrupted, it can be specified by the history pseudo-state. Then, sometimes it is very convenient to change the transition's destination state depending on some condition, for example, on the type of the received message. Conditional pseudo-states and pseudo-transitions (Fig. 4. right) serve for that purpose.

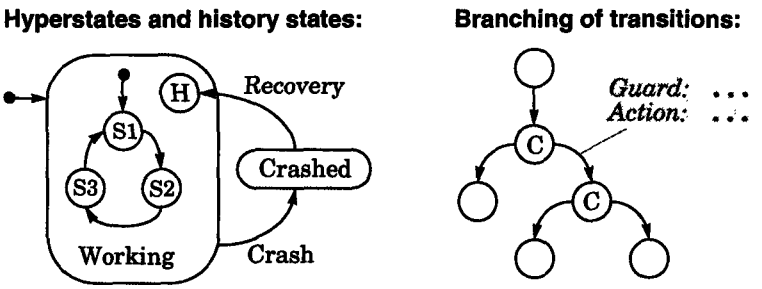


Fig. 4. Behavior. Extensions

As it was mentioned before, all the data processing is specified in COVERS in C language. Actually, C is the *only* language the user has to be familiar with. The user can write a C module containing constants, type definitions and functions that will be shared by processes. Also, each process may have its private C module containing private variables,

functions, etc. Of course, there is no function like *main()*, since the control flow is defined by the state diagram of the process. Functions are executed only if they are called during the firing of transitions or checking the transition's guards.

The semantics of the modelling language is defined as follows. The TTS $\langle \Sigma, s^0, T, C, d \rangle$ is built on the basis of the concrete model. In this TTS

Σ contains all possible combinations of three components:
- current states of all processes,
- values of all private and shared variables,
- values of all cast input queues.

s^0 is the TTS initial state, where
- every process is in its initial state,
- private and shared variables are initialized,
- all the cast queues are empty.

T contains one TTS transition for:
- each transition of each process marked with the reception of the cast message. For those TTS states where this transition is not open, or the corresponding queue is empty, the next TTS state is *Null*. Otherwise the next TTS state is obtained by moving the message from the queue to the given private buffer, executing the action of the transition, and moving the process to the new state.
- each pair of transitions of two different processes marked with the matching rendezvous functions. For those TTS states where at least one of the transitions is not open, the next TTS state is *Null*. Otherwise the next TTS state is obtained by executing the data exchange function, executing actions of both transitions, and moving the processes to the new states.
- each transition of each process marked with the time delay. For those TTS states where this transition is not open, the next TTS state is *Null*. Otherwise the next TTS state is obtained by executing the action of the transition, and moving the process to the new state.

C contains all pairs $\langle \tilde{\tau}, \tau \rangle$ of TTS transitions where both $\tilde{\tau}$ and τ include the transitions of one process, and the hierarchy level of the process' transition included in $\tilde{\tau}$ is higher or equal to the hierarchy level of the process' transition included in τ.

d, the firing delay function,
- is 0 for all TTS transitions corresponding to a cast message reception or a to rendezvous.
- for the TTS transitions corresponding to the process' transitions marked with the time delay, is equal to this time delay value.

Now, the semantics of the concrete model is the set of all possible computations of the TTS built according to these rules. It is easy to check that all timing assumptions, and all explanations of the communication mechanisms and state machines we gave are nothing else but just the consequences of this formal semantics. The concrete model now becomes fully executable.

4 Analysis of real time temporal properties

Among the different analysis tools provided by COVERS, one of the most interesting is the capability of testing the real time temporal properties. The general idea is the one proposed in [H92]: the user constructs the "special piece of behavior" tuned to enter the error state whenever the violation of the desired property takes place. This object is executed in parallel with the main system model, it watches everything what happens in the system, but it cannot affect the system behavior. Such objects are called *spies*.

The transition guards in the spy state diagram may refer to every variable, private or shared, to firing of a particular transitions, to rendezvous, etc. The only type of event allowed there is a time delay, or an empty event - instant transition. The way the instant transitions are executed is a bit special. First, the next state of the main system model is obtained, and then the guards of the instant transitions are checked. If some of them evaluate to TRUE, the spy makes a step *synchronously* with the system.

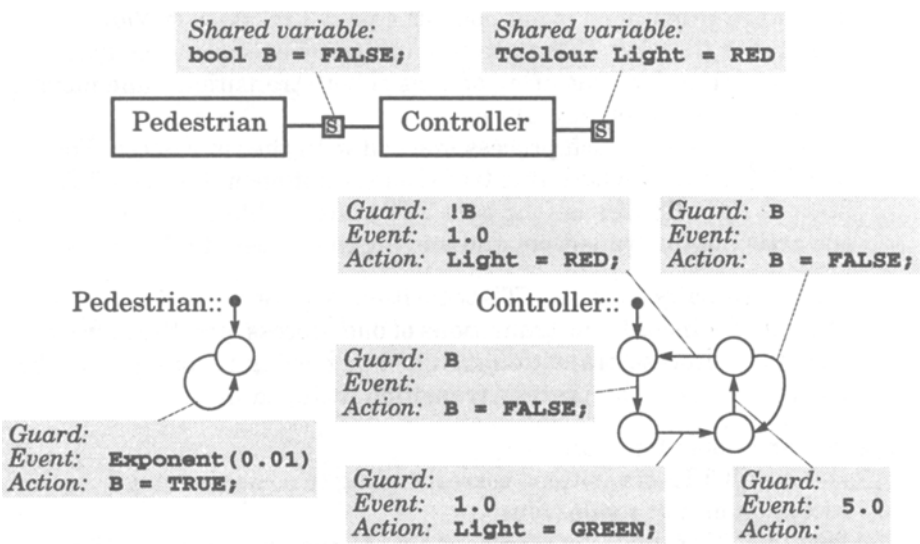


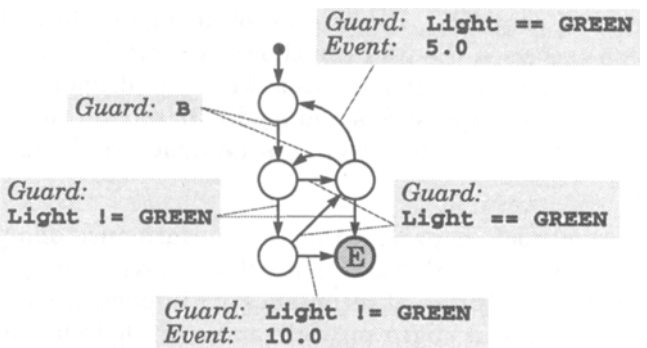
Fig. 5. Traffic lights control system

Let us consider the traffic lights control system example from [HMP92]. The model consists of two processes: the traffic light controller and the Poisson stream of pedestrians - see Fig. 5. The variable *Light* corresponds to the traffic light for the pedestrians. Pedestrians press the button once every 100 seconds, on average. When the button is pressed, the Boolean shared variable *B* is set to TRUE. If the *Light* is RED and *B* is set to TRUE, the *Controller* resets *B* and, in 1 second, shows the GREEN. After a 5 second pause, B is tested again. If it is still FALSE and remains FALSE for 1 second more, the RED light is shown. Otherwise, *B* is reset and *Controller* repeats the 5 second pause.

In the original paper the following two requirements are put upon the system:

1. Whenever *B* is TRUE then *Light* is GREEN within 5 seconds for at least 5 seconds.
2. Whenever *B* has been TRUE for 25 seconds, then *Light* is RED.

1. $\Box$ (**B** $\rightarrow$ $\Diamond_{[0,10]}$ ($\Box_{[0,5]}$ (**Light==GREEN**)))



2. $\Box$ ($\Box_{[0,25]}$ **!B** $\rightarrow$ $\Diamond_{[25,25]}$ (**Light==RED**))

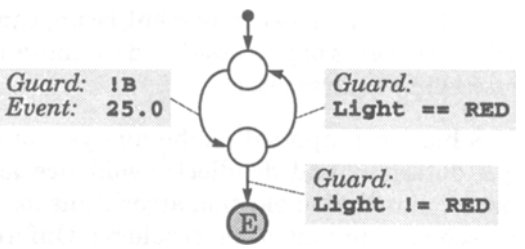


Fig. 6. Spies and temporal formulas expressing the requirements

The corresponding bounded temporal operators notation (see [AH92] for the details) and spies for these requirements are shown in Fig. 6.

When the *B* becomes TRUE, the first spy comes to the state *Check* and tests the *Light*. If it is already GREEN, it comes to the state *Green*, if not - to the state *Red*. If while the spy is in the state *Red*, *Light* does not become GREEN within 10 seconds, or if while it is in the state *Green* the *Light* becomes *Red*, the error state is entered. The second spy is straightforward.

Thus, the spies transform the temporal property into the reachability property, which can be easily tested during the system execution. If we manage to visit all the reachable states of the system model, and spy never enters the error state, the corresponding temporal property holds. But the most important thing about spies is that even if we cannot construct the whole state space, they will still *test* the property along all the simulation runs.

The two spies above were constructed manually. The subclass of the real time temporal properties which can be represented by spies, and the algorithm of spy construction still are to be found.

5 Other functionality of COVERS and conclusion

COVERS runs in the MS Windows environment and provides a convenient and easy GUI. It has a lot of debugging facilities. During the step-by-step execution the user can choose among the available alternative steps of the system, can put watches or break conditions on any variable or event, can evaluate expressions and modify data, and so on. All debugging is done in the terms of source languages, i.e structure charts, state diagrams and C code.

A number of tools is provided for the performance analysis. There is a library of standard distributions, special data structures and functions for statistics collection. Different techniques are implemented for the collection of statistics relating to migrating data and relating to particular processes, states or other static objects.

The user can specify a type of process, can organize processes hierarchically, and, what is the most important issue, can specify an array or any other periodic structures of processes and communication objects, the size of the structure being a parameter.

So far, COVERS has been applied to the analysis of several multicast protocols, deadlock detection and deadlock avoidance algorithms for the distributed databases, distributed election algorithms for the fault-tolerant systems, real-time scheduling and other problems. Unfortunately, because of the limited size of the paper, we can not afford to give here any of these interesting case studies.

References

[AH92] R. Alur and T.A. Henzinger. *Logics and Models of Real Time. A Survey*. LNCS, 1992.

[BONeS] *BONeS Designer. Modelling Reference Guide. Version 2.6.* Comdisco Systems, a business unit of Cadence Design Systems, Inc. USA, 1993.

[DCPN] *Design/CPN. A Reference Manual. Version 1.75.* Meta Software Corporation, USA, 1991.

[H87] D. Harel. *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming, Vol. 8, No. 3, June 1987, pp 231-274.

[HLNPPSST90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring and M. Trakhtenbrot. *STATEMATE: A Working Environment for the Development of Complex Reactive Systems.* IEEE Transactions on Software Engineering, Vol. 16, No. 4, April 1990, pp 403-414.

[HMP92] T.A. Henzinger, Z. Manna and A. Pnueli. *Timed Transition Systems*. Technical Report TR 92-1263, Department of Computer Science, Cornell University, January 1992.

[P92] A.S. Palmer. *An Illustration of Data Resources + Transaction Modelling as Applied to a Simple Network Problem*. Scientific and Engineering Software Inc., USA, March 1992.

[SES] *SES Workbench. User's manual*. Release 2.1. Scientific and Engineering Software Inc., USA, 1992.