Bridging The Gap Between The Design and Implementation of Hard Real-Time Systems

Homayoun Dayani-Fard
David Alex Lamb
{dayani,dalamb}@qucis.queensu.ca

September 3, 1996
External Technical Report
ISSN-0836-0227-
96-397

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Document prepared September 3, 1996

Abstract

There exists a gap between the design and implementation of hard real-time systems. During the design stage, few assumptions are made about the underlying execution environment; during the implementation stage, intimate knowledge of the underlying execution environment is required. To narrow this gap, we propose a phased approach to the design and implementation of hard real-time systems based on Timed CSP. During the first phase, logical design, we make no assumptions about the execution environment. Later, in the second phase, we construct a model of the target execution environment in Timed CSP. The logical design, from the first phase, is then transformed to conform to the model of the target execution environment.

1 Introduction

Hard real-time systems have stringent timing constraints which must be satisfied under all circumstances. Typically, the timing requirements are dealt with at later stages of the implementation. The main justification for this delay is that timing constraints are primarily due to the physical limitations of the underlying execution environment. To reason about the timing characteristics of a system, one needs intimate knowledge of the underlying computer system.

In recent years, formal methods have been used as a means to increase the reliability of real-time systems [6]. New languages have been specifically designed for specifying and reasoning about timing characteristics of real-time systems. However, despite their success, there still remains a gap between the design and implementation of real-time systems. Most formal methods ignore the underlying execution environment. Typically, during the design stage few assumptions are made about the underlying execution environment, whereas during the implementation, intimate knowledge of the underlying execution environment is needed. As a result, to increase the level of confidence in hard real-time systems, we must make explicit assumptions about the underlying execution environment [9].

This paper demonstrates a mechanism to narrow the gap between the design and implementation of hard real-time systems. To be able to formally reason about the behavior of a hard real-time system, we must include the main characteristics of the target execution environment into our formal notation. In our approach, we construct a formal model of the execution of a program as a parallel composition of two Timed CSP processes; one representing the application program and the other representing the underlying processor. Using the model of the underlying execution environment, we can divide the task of designing a real-time system into three phases: *logical design, physical design,* and *implementation* [5]. During the logical design phase, the designer assumes an ideal execution environment and constructs a set of Timed CSP processes that satisfies the specifications of the system. (This is according to the approach proposed by Davies, Jackson, and Schneider [3, 4, 8].) In the physical design phase, the processes defined in the logical design are transformed to processes which conform to the model of the underlying execution environment, scheduling processes are constructed, and execution time of tasks are measured or estimated. In the final stage, the physical design processes are transformed to program segments which can be executed on the system.

This approach separates the initial design from the implementation in a systematic way. After the initial design we can iterate between the intermediate stage and the implementation until a solution can be shown to be consistent with the specification. If the behavior specification of the system is represented as $\mathcal{F}$, a set of predicates on the observations of the system, and the system as a set of Timed CSP processes Q, the proof obligation of the

logical design phase is to show

$$Q \textbf{ sat } \mathcal{F}$$

During the physical design stage, the proof obligation is to show that the model of the execution environment, including the application processes, R, refines the logical model of the system Q:

$$R \sqsupseteq Q$$

This approach enables the designer to have more confidence in the final implementation. Further, depending on the level of formality required, the model of the underlying execution environment can have more detail. Refinement of specifications to implementations is well known; what is novel about our approach is the explicit introduction of a formal representation of the execution environment.

In section 2 we introduce our approach to modeling the underlying execution environment by constructing an ideal processor. In section 3 we construct a model of a simple processor and identify the key assumptions of the ideal execution environment. We then show how to construct a simple model of an example execution environment, RNet [1], and show how a watchdog timer can be implemented for this environment. Other execution environments can similarly be modeled using the approach presented in this paper.

We assume the reader is familiar with the basic concepts of Timed CSP [2]. The definition of the operators used in this paper are included in Appendix A.

2 Modeling An Ideal Processor

The computational model of Timed CSP [2] assumes that each process executes on its own processor. Further, the execution of each event takes zero time. In this section we begin introducing our modeling approach by making the computational model of Timed CSP explicit. That is, every Timed CSP process executes on its own ideal processor. The ideal processor, as we will see, is the identity process, hence the semantics of Timed CSP remains unchanged.

Consider a simple processor P which can repeatedly perform one of the three operations a, b, or c. We can model this processor as a Timed CSP process described by:

$$P \;=\; (a \rightarrow P) \;\Box\; (b \rightarrow P) \;\Box\; (c \rightarrow P)$$

Informally, our simple processor P offers its environment (the program) the choice among the events (operations) a, b and c. Next, consider a simple program Q which specifies the execution of operation a, followed by the operation

b and successful termination. This program can be described by the Timed CSP process

$$Q = a \rightarrow b \rightarrow SKIP$$

We define the execution of program Q on processor P as the lockstep synchronization of the respective processes, i.e., $Q \mid\mid P$. The parallel composition is consistent with our view of program execution because $Q \parallel P = Q$. The functional behavior of the processor is identical to that specified by the program. If the program specifies an operation that the processor cannot perform, the outcome will be deadlock. In other words, our notion of *executability* can be defined as follows: a program Q is executable on the processor P if and only if the set of events in its description is a subset of those events in the description of the processor P. Formally, if αP is the set of events in the description of P and αQ is the set of events in the description of Q, Q is executable on processor P if and only if $\alpha Q \subseteq \alpha P$.

We generalize our simple processor P to a processor which can perform any operation x drawn from the set A

$$GP = (x : A \rightarrow GP)$$

Further, an ideal processor IP can be described by extending the set A to the universal alphabet of events. If Σ is the universal alphabet of events, then the ideal processor IP can be described as

$$IP = (x : \Sigma \rightarrow IP)$$

The ideal processor can perform any event x, drawn from the universal alphabet, on behalf of the program in zero time.

The computational model of Timed CSP can now be rephrased as: each process executes on its own ideal processor. This statement is justified since the ideal processor is in fact the process RUN, as described by Hoare [7], and is the identity process for parallel composition:

$$Q \mid\mid IP = Q$$

During the initial design we can assume an ideal execution environment where each process executes on its own processor and execution times are zero. In effect, during the initial design we can ignore the underlying execution environment.

3 Modeling A Simple Processor

The ideal processor described in the previous section was modeled according the computational model of Timed CSP. The computational model assumes

that (1) each process executes on its own processor and (2) each operation takes zero time to execute. In this section we first remove the second assumption of the computational model of Timed CSP by modifying the ideal processor such that each operation takes a finite non-zero time. Next we describe how the first assumption of the computational model of Timed CSP can be removed. In the next section we construct a simple model of a particular execution environment, namely RNet, and show how the first assumption can be removed.

Our simple timed processor TP is willing to perform any operation according to its program except each event takes non-zero time units. Formally, if t_i is the time associated with an operation x_i drawn from the universal alphabet, our simple processor can be described by

$$TP = (x_i : \Sigma \stackrel{t_i}{\rightarrow} TP)$$

The difference between processors TP and IP is that processor TP, after performing an operation x_i, will not be ready to perform another operation until t_i time units has elapsed. On the other hand, the processor IP will immediately be ready to perform another operation. For example, the execution of process Q, described in the previous section, on the processor TP may result in a timed trace

$$\langle (a, 0), (b, t_a), (\surd, t_a + t_b) \rangle$$

where t_a and t_b are the delays associated with the execution of the operation a and b respectively. Thus, we observe the occurrence of a at time 0, b at time t_a, and successful termination at time $t_a + t_b$.

The first assumption of the computational model of Timed CSP concerns the number of processors in the system: each process executes on its own processor. In real life situations this assumption is generally inaccurate; two or more processes may execute on the same processor. In this paper, we ignore the possibility of asynchronous communication among processes. We do not allow communication between two processes that are executing on the same processor. Two processes that need to communicate with one another must be assigned to different processors. Note that this assumption can be removed by modeling the asynchronous communication mechanism provided by the execution environment.

For example, consider two simple processes Q_1 and Q_2 which must execute on the same processor:

$$Q_1 = a \rightarrow b \rightarrow Q_1$$

$$Q_2 = c \rightarrow d \rightarrow Q_2$$

The interleaved execution of these two processes on processor TP can be described by

$$(Q_1 \;|||\; Q_2) \;||\; TP$$

The pattern of execution, in this case, is a sequence of events drawn from the set $\{a, b, c, d\}$ and the order of events is nondeterministic.

As Xu and Parnas [10] argue, to guarantee that a hard real-time system satisfies its timing constraints, static scheduling must be used. In other words, the execution pattern of processes must be predetermined. Assuming static scheduling for our processor, we can determine an exact pattern of execution. For example, we may specify that the pattern of execution of the processes Q_1 and Q_2 to be

$$\langle acdbcdacdb \cdots \rangle$$

To be able to model such execution pattern, we must resolve the interleaved execution of processes. Therefore, we need to introduce a notion of scheduling. We model a process, namely the scheduler, which forces the execution pattern of concurrent processes to follow the predetermined pattern. In the next section, we construct a simple model of RNet and show how the scheduling can be resolved.

4 A Model of RNet

RNet [1] is a distributed real-time system consisting of five homogeneous processors, called *nodes*, which are interconnected by a local area network . Each node has its own copy of the kernel which provides scheduling and message passing facilities to the application processes assigned to that node. RNet uses static scheduling and hence the execution time of all "tasks" must be known in advance or worst case estimation of the execution time must be provided.

A real-time program in RNet consists of a set of concurrent processes which communicate among each other by message passing. Each process is structured as a sequence of one or more tasks. A task is an execution stream with a deadline and an estimated execution time. The period of a process is the period of all of its tasks. Tasks are scheduled based on their deadlines (i.e. earliest-deadline-first). Upon completion, each task must send a message to the scheduler and report the completion of its execution.

4.1 A Model of an RNet Node

In RNet, tasks are schedulable units and each process is divided into one or more tasks. As mentioned previously, tasks must voluntarily release the processor by sending a message to the scheduler. Following the RNet definitions, we divide processes, those representing the application programs, into tasks. Each task can be considered a sequential process, in particular, one that successfully terminates. For example, we can divide our processes Q_1 and Q_2 (as described in section 3) into tasks as follows:

$$Q_1' = T_{11}; T_{12}; Q_1'$$

$$Q_2' = T_{21}; Q_2'$$

where tasks T_{ij}'s are sequential processes defined as

$$T_{11} = dispatch_1 \rightarrow a \rightarrow trap_1 \rightarrow SKIP$$

$$T_{12} = dispatch_1 \rightarrow b \rightarrow trap_1 \rightarrow SKIP$$

$$T_{21} = dispatch_2 \rightarrow c \rightarrow d \rightarrow trap_2 \rightarrow SKIP$$

The event $trap_i$ corresponds to sending a message to the scheduler signaling the completion of the task, and the event $dispatch_i$ represents the start of the execution of a task. A scheduling process for our processor can now be described as

$$S = dispatch_1 \rightarrow trap_1 \rightarrow dispatch_2 \rightarrow trap_2 \rightarrow S$$

The execution of processes Q_1 and Q_2 on the processor TP must proceed under the control of the scheduling process S. This execution can be described by

$$S \; [[\mathcal{S}]] \; ((Q_1' \; ||| \; Q_2') \; || \; TP)$$

where $\mathcal{S} = \{dispatch_1, dispatch_2, trap_1, trap_2\}$ is the set of scheduling events. Thus, processes Q_1' and Q_2' execute concurrently on processor TP and the execution proceeds under the control of the scheduling process S.

The last feature that we model for RNet is the inter-node message-passing facility. RNet has two message passing primitives: `SEND` and `RECEIVE`. The primitive `SEND` is always non-blocking, whereas `RECEIVE` can be blocking or non-blocking. The blocking `receive` has a timeout value associated with it. When a process requests a blocking `receive`, its execution is paused until a message arrives or it times out, in which case the process continues execution following the `receive` primitive.

To model the time out feature of blocking `receive`, we need to provide an interrupt mechanism. In RNet, the scheduler uses a timer to generate interrupt. Our interruptible model of the processor can be described as

$$ITP \mathrel{\widehat{=}} TP \; \triangle \; (i \rightarrow ITP)$$

where event i represents the special event *interrupt*. If the processor is executing a blocking Receive, the occurrence of the event i will cause the processor to restart. In other words, the processor ITP behaves like the processor TP until the event i becomes available, which causes the processor to behave like $(i \rightarrow ITP)$. To avoid unnecessary complexity by modeling a timer process, our model allows the scheduler to generate interrupts. Further, we modify the description of processes which have a blocking `receive` such that they can be

interrupted by the scheduling process. For example, consider the process Q_3 described as

$$Q_3 \;=\; a \rightarrow \begin{array}{l} b \rightarrow c \rightarrow Q_3 \\ \overset{t_1}{\triangleright} \\ d \rightarrow Q_3 \end{array}$$

This process engages in the event a and then event b; if the event b does not occur within t_1, the control is passed to the process $d \rightarrow Q_3$. Let us assume that event b is a blocking Receive. We model the execution of this process as follows: we convert Q_3 to a process with one task, substitute the event b by an interruptible process $(b \triangle\ i)$, and remove the timeout value. The timeout value is used in the description of the scheduler. Formally, this is described by

$$Q_3' \;=\; dispatch_1 \rightarrow a \rightarrow \begin{array}{l} b \rightarrow c \rightarrow trap_1 \rightarrow Q_3' \\ \triangle \\ i \rightarrow d \rightarrow trap_1 \rightarrow Q_3' \end{array}$$

$$S = dispatch_1 \rightarrow (trap_1 \overset{t_1}{\triangleright} i \rightarrow trap_1); S$$

The execution of process Q_3 can now be modeled by

$$S \;|[\mathcal{S}]|\; (Q_3' \;||\; ITP)$$

where $\mathcal{S} = \{dispatch_1, trap_1, i\}$ is the set of scheduling events.

To summarize, a node of RNet can execute one or more (application) processes which do not communicate among each other. The application processes must be divided into schedulable units or tasks. All timing requirements must be removed from the description of the process, timeout constructs must be replaced by interruptible processes, and a scheduler process must be constructed according to the timing information.

More formally, an application process can be described as a sequential composition of one or more tasks:

$$Q = T_1; T_2; \cdots; T_n; Q$$

where each task T_j contains the events *trap* and *dispatch*

$$T_j = dispatch_j \rightarrow R \rightarrow trap_j \rightarrow SKIP$$

and R is a Timed CSP sequential process, in which all timing information is removed. All application processes executing on a node of RNet can be described by interleaving their respective processes:

$$Applications = \|_k \;\; Q_k \qquad k \in \mathbb{N}$$

The scheduling process repeatedly dispatches the application processes and waits for their completion or until the application processes time out. The scheduler has the form

$$\begin{aligned} S \;=\; & \mathit{dispatch}_1 \rightarrow (\mathit{trap}_1 \overset{t_1}{\triangleright} i \rightarrow \mathit{trap}_1); \\ & \vdots \\ & \mathit{dispatch}_m \rightarrow (\mathit{trap}_m \overset{t_m}{\triangleright} i \rightarrow \mathit{trap}_m); S \end{aligned}$$

where $t_l \in \mathbb{R}^+$. A node with all of its application processes can be modeled as

$$\mathit{RNet\text{-}Node} \mathrel{\widehat{=}} (S \;[\![\mathcal{S}]\!]\; (\mathit{Applications} \;||\; \mathit{ITP})) \setminus \mathcal{S}$$

In other words, the execution of application processes proceeds under the control of the scheduler.

4.2 A Simple Model of RNet

As noted earlier, RNet consists of five homogeneous processors which are interconnected by a local area network. To complete our model of RNet, we need to model the network connecting these processors. This can be accomplished by modeling the network as a Timed CSP process. However, for purposes of this paper, we model the connection between two nodes by the communicating parallel composition of the nodes. Let $\mathcal{I}$ be the set of events representing communication between two nodes of RNet, R_1 and R_2. We can described the model of RNet by

$$\mathit{RNet} = R_1 \;[\![\mathcal{I}]\!]\; R_2$$

Each node executes its own application processes and communicates with the other node through the set $\mathcal{I}$. Alternatively, we could model the communication of two nodes by using Timed CSP communication channels. In this case, we would also have to include the channel operations in the definition of the processor *ITP*. In the next section we show how this model of RNet can be used to design a watchdog timer.

5 An Example: A Watchdog Timer

To demonstrate the RNet model, we model the watchdog timer example described by Davies and Schneider [4]. A watchdog timer is a simple device for monitoring the activity of a component. Periodically, the component sends a signal to the timer to confirm that normal activity is taking place. If the component does not send the signal, indicating normal activity, within time t, the timer sound the alarm. The monitored component sends a signal on

the channel *reset* and alarm is sounded on the channel *alarm*. The watch dog timer [4] can be described by

$$\begin{aligned} Timer &= reset \xrightarrow{\tau_1} Timer \\ &\overset{t}{\triangleright} \\ &alarm \xrightarrow{\tau_2} STOP \end{aligned}$$

The collator process accepts input from the watchdog timer on the alarm channel. If the timer sounds the alarm, the collator stops the system. The collator process can be described by

$$Collator = alarm \xrightarrow{\tau_2} sound\text{-}alarm \xrightarrow{\tau_3} STOP$$

The *Watchdog* process is the communicating parallel composition of the collator and timer processes as described by

$$Watchdog = Timer \,|[alarm]|\, Collator$$

The description of *Watchdog* given by Davies and Schneider assumes ideal execution environment. Suppose we want to execute this system on RNet. In other words, our task is the second phase or physical design. We transform the given description of the distributed watchdog timer according the the RNet model described in section 4. We assign each of the processes *Timer* and *Collator* to one node of RNet. Each process is constructed as one task. We remove timing information in the description of *Timer* and replace the timeout construct by the interrupt mechanism. The description of the process *Timer* according to RNet, called T, can be described by

$$\begin{aligned} T &= dispatch_1 \rightarrow reset \rightarrow trap_1 \rightarrow T \\ &\triangle \\ &i_1 \rightarrow alarm \rightarrow trap_1 \rightarrow STOP \end{aligned}$$

The scheduling process for node 1, where process T executes, is represented as

$$S_1 = dispatch_1 \rightarrow (trap_1 \overset{t}{\triangleright} i \rightarrow trap_1); S_1$$

The model of the processor for each node can be described by

$$\begin{aligned} ITP_k &= TP_k \triangle (i_k \rightarrow ITP_k) \\ TP_k &= reset \xrightarrow{\tau_1} ITP_k \\ &\quad \Box\ sound\text{-}alarm \xrightarrow{\tau_3} ITP_k \\ &\quad \Box\ alarm \xrightarrow{\tau_2} ITP_k \\ &\quad \Box\ dispatch_k \rightarrow ITP_k \\ &\quad \Box\ trap_k \rightarrow ITP_k \end{aligned}$$

where τ_j is the time associated with execution of each operation and $k = 1, 2$. The model of node 1, R_1, can be described by

$$R_1 \quad = \quad (S_1 \; |[\mathcal{S}_1]| \; (T \; || \; ITP_1)) \setminus \mathcal{S}_1$$

where $\mathcal{S}_1 = \{dispatch_1, trap_1, i_1\}$ is the set of scheduling events.

Similarly the RNet definition of the *Collator* process can be described by a single task process C

$$C \quad = \quad dispatch_2 \rightarrow alarm \rightarrow sound\text{-}alarm \rightarrow trap_2 \rightarrow STOP$$

The corresponding scheduling process S_2 can be constructed as

$$S_2 \quad = \quad dispatch_2 \rightarrow trap_2 \rightarrow STOP$$

The model of node 2 of RNet, R_2, can be described by

$$\begin{array}{rcl} R_2 & = & (S_2 \; |[\mathcal{S}_2]| \; (T \; || \; ITP_2)) \setminus \mathcal{S}_2 \\ \mathcal{S}_2 & = & \{dispatch_2, trap_2, i_2\} \end{array}$$

The RNet model of the *Watchdog* can be now be described by combining the two nodes via a communicating parallel composition:

$$RNet \quad = \quad R_1 \; |[alarm]| \; R_2$$

The next step is to show that the RNet model of the Watchdog, *RNet* refines the model of the ideal execution environment. Therefore, we must show that $RNet \sqsupseteq Watchdog$.

Further, we can repeat the physical design stage and make the model the of execution environment more detailed, measure the exact timing of the tasks, estimate the worst case execution times, and so forth. As a result, the design process becomes iterative where at each stage we bring more implementation detail to our model and demonstrate that the current model refines the previous one.

6 Discussion

To raise the level of confidence in hard real-time systems, the characteristics of the underlying execution environment must be included in the design stage as early as possible. However, most formal methods used for real-time systems ignore the underlying execution environment. As a result, there remains a gap between the design and implementation of hard real-time systems. During the design stage few assumptions are made about the execution environment, whereas during the implementation intimate knowledge of the execution environment is required.

In this paper we presented an approach based on Timed CSP [2] to bridge this gap.

6.1 Contribution

We showed how program execution can be modeled as parallel composition of two Timed CSP processes representing the program and its underlying processor. Further, we demonstrated the feasibility of this approach by constructing a simple model of a particular execution environment, RNet, using this approach. Lastly, we demonstrated how the watchdog timer example can be refined to be in conformance with RNet.

Our design approach has three phases. During the first phase, logical design, we assume an ideal execution environment. For the second phase, physical design, we need to construct a model of the target execution environment in Timed CSP and transform the solution of the logical design to conform to the model of the target execution environment. In the third phase, implementation, we convert the physical design to program modules executable on the target environment. Each stage has its own proof obligations [5].

6.2 Future Work

As any approach to design and implementation of software systems, further use of our approach requires an initial investment. We need to construct models of the execution environments that are going to be used as a target for hard real-time systems. In particular, embedded systems where the development environment is usually different from the execution environment can benefit from this approach. The initial investment is to construct a model of the development environment but the model can be reused for different projects.

Large-scale use of the approach requires tools that can automate some of the tasks involved in the design and implementation of the system. For example, in the RNet model, the construction of scheduling processes is currently semi-automated. The RNet timing analyzer tries to find a static scheduling that satisfies all timing requirements, if successful, the result can be used to define scheduling processes. The transformation of logical design processes to those complying to the model of the underlying execution environment is still not automated. This requires a more rigorous and formal definition of tasks and how a process, according to its specification, can be divided into tasks. Lastly, the use of model checkers and semi-automated theorem provers can ease the proof obligations of each stage.

References

[1] M. F. Coulas, G. H. MacEwen, and G. Marquis. RNet: A hard real-time distributed programming system. *IEEE Transactions on Computers*, C-36(8):917–932, August 1987.

[2] J. Davies. *Specification and Proof in Real-Time CSP.* Cambridge University Press, 1993.

[3] J. Davies, D. Jackson, and S. Schneider. Making Things Happen in Timed CSP. Technical Report PRG-TR-2-90, Oxford University, 1990.

[4] J. Davies and S. Schneider. Real-Time CSP. Technical report, Oxford University, 1994.

[5] H. Dayani-Fard. Development of a Real-Time System: A Formal Approach. Master's thesis, Queen's University, 1995.

[6] S. Gerhart, D. Craigen, and T. Ralston. Experience with formal methods in critical systems. *IEEE Software*, 21(1):21–39, January 1994.

[7] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall international, 1985.

[8] D. M. Jackson. The specification of aircraft engine control software using timed CSP. Technical Report PRG-TR-15-90, Oxford University, 1990.

[9] A. K. Mok. Coping with implementation dependencies in real-time system verification. *LNCS(600)*, 1991.

[10] J. Xu and D. L. Parnas. On satisfying timing constraints in hard-real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, January 1993.

A Timed CSP Operators

The description of Timed CSP operators used in this paper is given below. For a more comprehensive discussion of Timed CSP, see [2, 4].

Σ	Universal set of events
$\surd$	Event marking the successful termination
αP	Set of events in the description of P
$SKIP$	Successful termination
$STOP$	Deadlock process
$P \setminus A$	A process that behaves as P except that events from the set A are no longer visible to the environment.
$a \rightarrow P$	A process that engages in a then behaving as P
$P \Box\, Q$	(External) choice between P and Q
$x : A \rightarrow P$	Choice between events in the set A
$P; Q$	Sequential composition of P and Q
$P \mid\mid Q$	Parallel composition; P and Q must synchronize on events in $\alpha P \cap \alpha Q$
$P \;\|\!\|\!\|\; Q$	Interleave concurrency; P and Q evolve independently
$P \;[\![A]\!]\; Q$	Processes P and Q evolve independently but must cooperate on events from the set A
$P \overset{t}{\triangleright} Q$	A process initially prepared to behave as P but if no event has occurred within t time units, the process behaves as Q instead.
$P \triangle\; Q$	A process that behaves as P, but may be interrupted at any time by the initial event of Q.