Chapter 8

The Event Calculus

Formal Specification of Real Time Systems by means of Diagrams and Z schemas

The "Event Calculus" is a diagrammatic notation which provides an easily used means of formally specifying the behaviour of concurrent systems. It can describe synchronous and asynchronous communications, data flow modelling and function application, and the expression of temporal constraints. It has the ability to abbreviate the descriptions of complex state changes, such as data base updates, by means of Z schemas.

The Calculus models system states with sets of parameterised state machines which can communicate via n way synchronisations known as "events". In comparison with process algebras such as CSP, CCS and LOTOS, the calculus is relatively easy for a non specialist to use. It is also easier to present specifications that can be understood by developers who do not have a sophisticated mathematical background. The calculus provides a good control of the level of abstraction used in a model.

In this chapter we introduce the Event Calculus with a series of examples and give a formal mathematical interpretation of the diagram notation.

8.1 Introduction

The Event Calculus resembles process algebras such as CCS (Milner 1989) and CSP (Hoare 1985) in that the mathematical model of communication derives from a simultaneous state change in more than one state machine. We provide a model theoretic description of how the behaviour of a composite state machine can be derived from the behaviour of its component state machines. The basic model is then extended to include asynchronous events, value passing, function application and time. Finally, we introduce the use of supplementary Z schemas to augment the diagram notation and the use of "Schema Transitions" to describe complex state changes such as data base updates.

An important aim of the calculus is that the use of diagrams should give the user a more intuitive and direct view of system behaviour than can be achieved by algebraic expressions alone. Unlike less formal diagram notations, such as DFDs, our diagrams give a complete model of system behaviour and may be thought of as the user interface to an underlying algebra of machine behaviours. Unlike other formal models of concurrency which can also provide a diagrammatic representation of simple processes,

[0] **This is a chapter taken from the Ph.D. thesis "Practical and Theoretical Aspects of Forth Software Development". A full copy of the thesis is available from the British library. Both the this chapter and the thesis are Copyright © Peter Knaggs.**

the diagram notation for the Event Calculus is fully equipped to deal with complex examples involving parameter passing, function application, asynchronous events and data base updates.

The Event Calculus is particularly well suited for use with the FORTH language. Each FORTH actor (or task) can be represented in the Event Calculus as a state machine.

8.2 State Machines

The Event Calculus is based on state machines which are associated with a set of state changing "events". Our state machines are described in terms of a set of states, a set of events and a next state function which maps a state event pair to a new state.

Our first example consists of a pair of state machines, V and C. The next state functions ψV and ψC are as shown in figure 8.1.

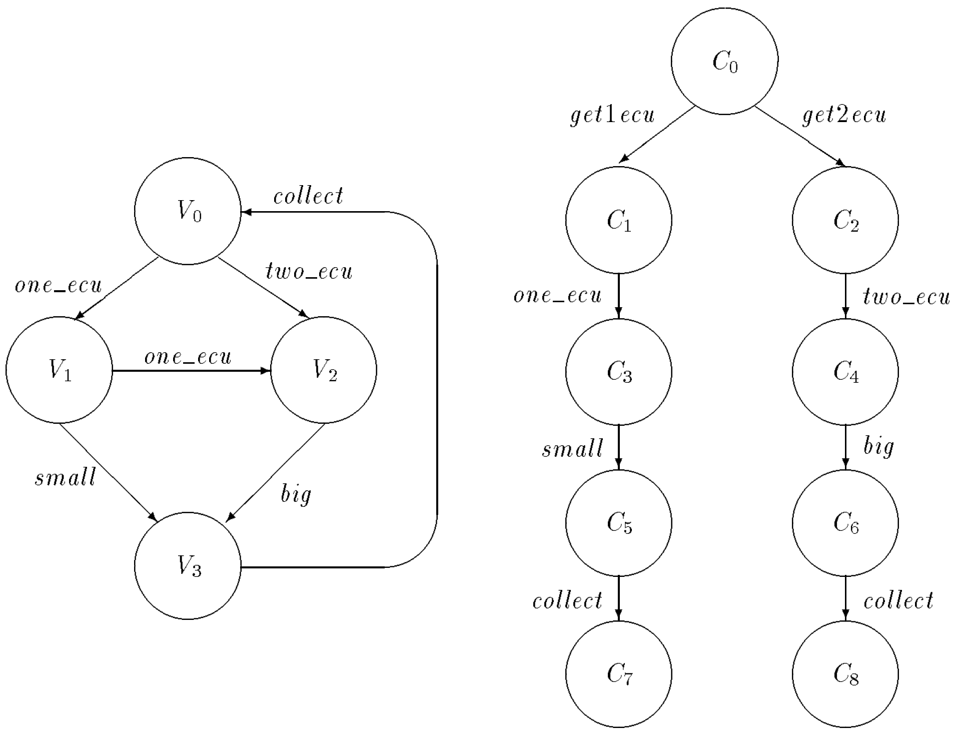


Figure 8.1: The state machines V and C

V is a simple futuristic vending machine which can accept one ecu or two ecu coins and can dispense either a small or big chocolate bar.

$$states\ V = \{V_0, V_1, V_2, V_3\}$$

$$events\ V = \{one\_ecu, two\_ecu, small, big, collect\}$$

C is a child who gets a one ecu coin or a two ecu coin and inserts it into the vending machine to purchase a chocolate bar.

$$states\ C = \{C_0, C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_8\}$$

$$events\ C = \{get1ecu, get2ecu, one\_ecu, two\_ecu, small, big, collect\}$$

Some of the events of C, such as *one_ecu*, are shared with V. When a shared event occurs both C and V will simultaneously move to a new state. We call such an event "synchronous". The child puts the coin in at the same instant as the machine has the coin put in.

We now describe (informally) the Event Calculus rules for determining the behaviour of the composite machine $\{V, C\}$.

Let us suppose we start in state $\{V_0, C_0\}$. In state V_0 machine V is ready to participate in events *one_ecu* and *two_ecu*. However, these are both events that are in the event set of machine C. Events are enabled only when all machines capable of taking part in them are ready to do so. Thus from our initial state these events are disabled and cannot occur.

In state C_0, machine C is ready to perform the events *get1ecu* and *get2ecu*. These events are unique to machine C, so they are enabled (We could also say they can occur because all machines capable of taking part in them are ready to do so, the only such machine being C).

At each stage, any enabled event can occur. Suppose *get1ecu* occurs. The composite machine is now in state $\{V_0, C_1\}$. We denote this state change by writing:

$$\{V_0, C_0\} \xrightarrow{get1ecu} \{V_0, C_1\}$$

Now both C and V are ready to take part in the event *one_ecu*. This is now the only event enabled. In fact the remainder of the behaviour of $\{V, C\}$ is deterministic. All possible composite behaviours of $\{V, C\}$ (there are only two) are shown in figure 8.2.

$$\begin{tikzcd}
 & \{ V_0, C_0 \} \arrow[dl, "get1ecu"'] \arrow[dr, "get2ecu"] & \\
\{ V_0, C_1 \} \arrow[d, "one\_ecu"'] & & \{ V_0, C_2 \} \arrow[d, "two\_ecu"] \\
\{ V_1, C_3 \} \arrow[d, "small"'] & & \{ V_2, C_4 \} \arrow[d, "big"] \\
\{ V_3, C_5 \} \arrow[d, "collect"'] & & \{ V_3, C_6 \} \arrow[d, "collect"] \\
\{ V_0, C_7 \} & & \{ V_0, C_8 \}
\end{tikzcd}$$

Figure 8.2: Graph of behaviours for the model of figure 8.1

We follow a naming convention that uses machine names derived from the common root of the state names for that machine. Thus the state names V_0, V_1, V_2 and V_3 have a common root V which is the name of the corresponding machine.

8.3 The Formal Model

We now provide formal rules for deriving the behaviour of a set of machines from the next state functions of each individual machine.

Basic types for our discussion are:

$$[MACHINE, STATE, EVENT]$$

$MACHINE$ is the set of state machines (or more precisely state machine identifiers).

$STATE$ is the set of possible machine states.

$EVENT$ is the set of events.

We define a "next state" function:

$$\psi : MACHINE \rightarrow ((STATE \times EVENT) \nrightarrow STATE)$$

The idea here is that ψ is a function from machines to next state functions. Thus ψm will return the next state function for machine m.

We define a function to identify the set of states associated with a particular machine and to specify that each state is associated with only one machine.

$$\begin{array}{|l}
states : MACHINE \rightarrow \mathbb{P}\, STATE \\
\hline
\forall\, m : MACHINE \bullet \\
\qquad states\ m = \mathrm{ran}(\psi m) \cup \mathrm{dom}\,(\mathrm{dom}\,(\psi m)) \\
\forall\, m_1, m_2 : MACHINE \bullet \\
\qquad m_1 \neq m_2 \Rightarrow states\ m_1 \cap states\ m_2 = \varnothing
\end{array}$$

We define a function to return the unique machine associated with a given state.

$$machine == \{s : STATE,\ m : MACHINE \mid s \in state\ m \bullet s \mapsto m\}$$

The basis of the Event Calculus is the definition of a function χ that describes the behaviour of a set of possibly communicating machines in terms of the next state functions of each individual machine. First, however, we define some functions and sets which we will use to make the definition of χ more readable.

First, we define a function that tells us whether an event can occur when a machine is in a given state.

$$\begin{array}{|l}
ready : STATE \times EVENT \rightarrow \mathbb{B} \\
\hline
\forall\, e : EVENT;\ s : STATE \bullet \\
\qquad ready(s, e) = (s, e) \in \mathrm{dom}\,(\psi\,(machine\ s))
\end{array}$$

Considering our vending machine V as an example:

$$\begin{array}{rl}
ready(V_0, one\_ecu) = \mathrm{true} & \text{as event } one\_ecu \text{ can occur in state } V_0. \\
ready(V_0, big) = \mathrm{false} & \text{as event } big \text{ cannot occur in state } V_0.
\end{array}$$

The *repertoire* of a machine is the set of all events in which it can participate. The set of *events* associated with the next state function of a machine is a subset of its *repertoire*[1].

[1] This distinction is useful because we may wish to provide a next state function which shows only part of the behaviour of a machine, a subset of *events* from a machines *repertoire*.

$$\begin{array}{|l}
events, repertoire : MACHINE \rightarrow \mathbb{P}\, EVENT \\
\hline
\forall\, m : MACHINE \bullet events\ m = \mathrm{ran}\,(\mathrm{dom}\,(\psi m)) \wedge events \subseteq repertoire
\end{array}$$

When considering a set of state machines we think of an event as causing a composite state change which may affect more than one machine. Valid composite states are ones in which each constituent state represents the state of a different machine. Formally, we define the set of valid state sets as follows:

$$\begin{array}{l}
validstateset = \\
\quad \{sset : \mathbb{P}\, STATE \mid \forall\, s_1, s_2 : sset \bullet \\
\qquad s_1 \neq s_2 \Rightarrow machine\ s_1 \neq machine\ s_2\}
\end{array}$$

In the context of a composite state set, an event may only occur if every machine which has that event in its *repertoire* is ready for it. We define a boolean function that will tell us whether a particular event is enabled for a given composite state.

$$\begin{array}{|l}
enabled : (validstateset \times EVENT) \rightarrow \mathbb{B} \\
\hline
\forall\, sset : validstateset;\ e : EVENT \bullet \\
\quad enabled(sset, e) = \\
\qquad (\forall\, s : sset \bullet e \in repertoire(machine\ s) \Rightarrow ready(s, e)) \\
\qquad \wedge\ (\exists\, s : sset \bullet ready(s, e))
\end{array}$$

We now define the function χ which derives the behaviour of a set of state machines from the individual behaviours of each machine plus the enabling rule for composite events. We will define χ to take a set of machines as its argument and to return a next state function for that set of machines. χ is described by giving its domain and by describing its application to an arbitrary element of its domain.

$$\begin{array}{|l}
\chi : \mathbb{P}\, MACHINE \rightarrow ((validstateset \times EVENT) \rightarrow validstateset) \\
\hline
\forall\, sset : validstateset;\ e : EVENT;\ mset : \mathbb{P}\, MACHINE \bullet \\
\quad \mathrm{dom}(\chi mset) = \{e : EVENT;\ sset : validstateset \mid \\
\qquad machine\ (\!|\ sset\ |\!) = mset \wedge enabled(sset, e) \bullet (sset, e)\} \\
\quad \wedge \\
\qquad (sset, e) \in \mathrm{dom}(\chi mset) \Rightarrow \\
\qquad\quad (\chi mset)(sset, e) = \{s' : STATE \mid \\
\qquad\qquad s' \in sset \wedge e \notin events(machine\ s') \\
\quad \vee \\
\qquad \exists\, s : sset \bullet e \in events(machine\ s) \wedge s' = (\psi(machine\ s))(s, e)\}
\end{array}$$

8.4 An Algebra of machine behaviours

The function χ maps from a set of machines to a function which describes the possible composite behaviours of those machines. We now introduce a binary operation to compose such behaviours. We write this operation as $||$ (pronounced "par").

$$\begin{array}{|l}
\_||\_ : range\ \chi \times range\ \chi \rightarrow \chi \\
\hline
\forall\, mset_1, mset_2 : \mathbb{P}\, MACHINES \bullet \\
\quad \chi mset_1 || \chi mset_2 = \chi(mset_1 \cup mset_2)
\end{array}$$

χ is now a homomorphism from the algebra of set unions $[\mathbb{P}\, MACHINES, \cup]$ to the algebra of machine behaviours $[range\ \chi, ||]$. It follows that $[range\ \chi, ||]$ is a commutative monoid (ie, $||$ is commutative, associative and has a unit element).

8.5 Labelled Transitions

For arbitrary $s, t : STATE;\ e : EVENT$ we introduce the notation:

$$s \xrightarrow{e} t$$

to indicate that the machine associated with state s goes from state s to state t when event e occurs. We call this a "labelled transition". Formally the notation is introduced as follows.

$$\_ \overset{-}{\Leftrightarrow\!\!\rightarrow} \_ : STATE \times EVENT \times STATE \rightarrow \mathbb{B}$$

$$s \overset{e}{\Leftrightarrow\!\!\rightarrow} t \Leftrightarrow (\psi(machine\ s))(s, e) = t$$

8.6 Simple Examples

In this section we provide some simple examples of Event Calculus models and at the same time introduce some additional notations for use in Event Calculus diagrams.

8.6.1 The specification of mutual exclusion (without fairness)

Consider two processes A and B, and a semaphore S.

$$states\ A = \{A_0, A_1\}$$
$$states\ B = \{B_0, B_1\}$$
$$states\ S = \{S_0, S_1\}$$

A_1 and B_1 are the critical regions of processes A and B.

S will be in state S_1 when one of the processes is inside its critical region and in state S_0 otherwise.

Starting from initial composite state $\{A_0, B_0, S_0\}$, any composite state which includes $\{A_1, B_1\}$ will be impossible to reach.

The individual machines are as shown in figure 8.3.

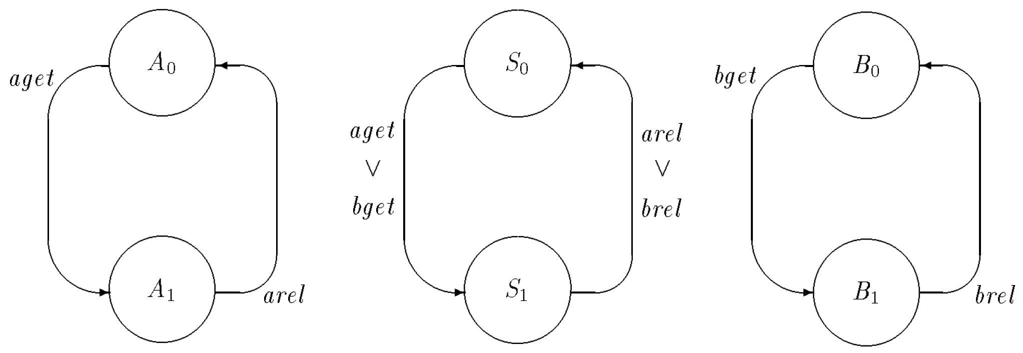


Figure 8.3: Mutual exclusion without fairness

This diagram introduces a convention by which two alternative transitions with the same start and end states are shown by a single line with an appropriate label. For example the arc from S_0 to S_1 which is labelled $aget \vee bget$.

$$\begin{tikzcd}
& \{ A_0, S_0, B_0 \} \arrow[dl, "aget"'] \arrow[dr, "bget"] & \\
\{ A_1, S_1, B_0 \} \arrow[ur, bend left, "arel"] & & \{ A_0, S_1, B_1 \} \arrow[ul, bend right, "brel"']
\end{tikzcd}$$

Figure 8.4: Graph of behaviours for the model given in figure 8.3

Suppose we start the model in composite state $\{A_0, B_0, S_0\}$. The possible transitions for the composite state are as shown in figure 8.4.

When two (or more) alternative transitions are available (in this case *aget* and *bget*) it is upto the designer to fire and trace the events as appropriate (see the trace on page 77 of section 8.6.4 for an example of tracing events).

If we were to give machine A precedence over machine B, we would observe the unfair nature of this model, in that machine B will *never* be capable of reaching state B_1. The event *bget* is only valid when we have the composite state $\{A_0, B_0, S_0\}$, however the event *aget* is also valid in this state. As we are favouring machine A over machine B, we now fire the *aget* event, thus preventing the *bget* event from ever being fired (a condition referred to as *indefinite postponement*).

8.6.2 Asynchronous Events

Machine A broadcasts job requests to machines B and C. The broadcast event, which we denote by *job*, is to be asynchronous. That is, B and C do not have to be ready for the broadcast for it to occur. We distinguish such asynchronous events in our diagram notation by underlining them in the graph of the state machine which originates them.

In this simple model, A knows that it can have two jobs outstanding at any one time and will not attempt to broadcast a further job until one of these is done (see figure 8.5).

Now consider the following composite state transitions from an initial composite state $\{A_0, B_0, C_0\}$

$$\{A_0, B_0, C_0\} \xrightarrow{job} \{A_1, B_1, C_1\} \tag{8.1}$$

$$\{A_1, B_1, C_1\} \xrightarrow{back} \{A_2, B_2, C_0\} \tag{8.2}$$

$$\{A_2, B_2, C_0\} \xrightarrow{job} \{A_3, B_2, C_1\} \tag{8.3}$$

$$\{A_3, B_2, C_1\} \xrightarrow{cack} \{A_4, B_2, C_2\} \tag{8.4}$$

According to our laws for composite transitions, the composite transitions 8.3 and 8.4 cannot occur, as they are not synchronised. To allow asynchronous events to be declared we must add some additional structure to our calculus.

One possibility would be to distinguish two types of event (synchronous and asynchronous) and formulate rules for the underlying calculus accordingly. Another approach, which we follow here, is to describe asynchronous events in terms of the existing calculus. In terms of the underlying formalism, the effect of declaring *job* as an asynchronous event originated by machine A is to add some additional null transitions to B and C to allow *job* to occur in all states of B and C.

Let the original next state function as given by the above graphs be ψ_0. We construct ψ_1 which

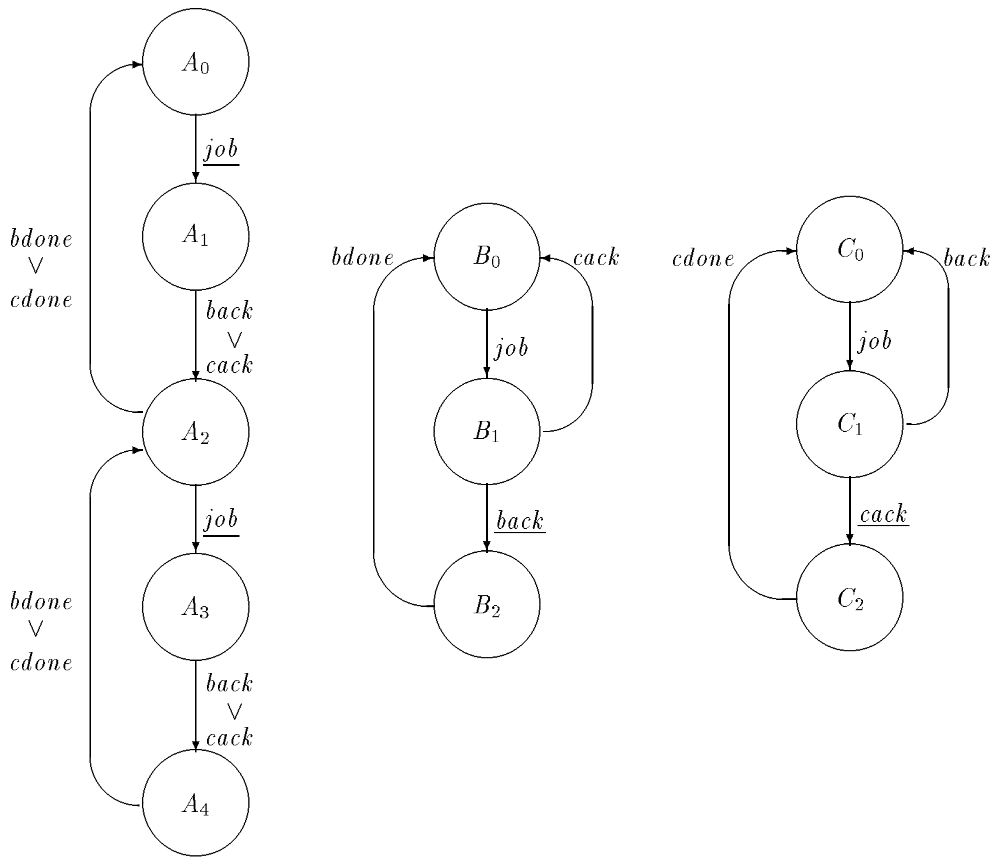


Figure 8.5: An example using asynchronous broadcasts

allows *job* to be asynchronously originated by A:

$$\psi_1 m = \\ \quad \text{if } m = A \text{ then } \psi_0 A \\ \quad \text{else} \bigcup_{s \in states\ m} \{(s, job) \mapsto s\} \oplus \psi_0$$

The idea is to add labelled transitions of the form $s \xrightarrow{job} s$ to the next state functions of B and C for all states s which are not otherwise ready to participate in the event *job*.

The event *back* is asynchronously originated by B and event *cack* is asynchronously originated by C. We can construct appropriate next state functions to express this as follows.

From ψ_1 we can construct ψ_2 which allows *back* to be asynchronously originated by B and from ψ_2 construct ψ_3 which allows *cack* to be asynchronously originated by C.

In general, we describe a constructor function *async* which takes a next state function ψ, a machine m^*, an event e and returns a new next state function $async(\psi, m^*, e)$ which has the additional labelled transitions required to allow e to be asynchronously originated by m^*.

$$SIMPLE\_NEXT\_STATE\_FUNCTION == \\ \qquad MACHINE \rightarrow (STATE \times EVENT) \nrightarrow STATE$$

$$\begin{array}{|l}
async : SIMPLE\_NEXT\_STATE\_FUNCTION \times MACHINE \times EVENT \\
\qquad \nrightarrow SIMPLE\_NEXT\_STATE\_FUNCTION \\
\hline
\forall\, \psi : SIMPLE\_NEXT\_STATE\_FUNCTION; \\
m, m^* : MACHINE; \\
e : events\ m^* \bullet \\
\quad e \in events\ m \wedge \\
\quad async(\psi, m^*, e)m = \\
\qquad \text{if } m = m^* \text{ then } \psi m \\
\qquad \text{else } \bigcup_{s : states\ m} \{(s,e) \mapsto s\} \oplus \psi m \\
\quad \vee \\
\quad e \notin events\ m \wedge \\
\qquad async(\psi, m^*, e)m = \psi m
\end{array}$$

8.6.3 Value passing

Our calculus is based on a primitive notion of synchronised events which do not, of themselves, admit any notion of direction in communication. However, we can build such a notion and use it to express ideas such as value passing and function application. The basic idea is taken from the value passing calculus of Ccs.

Consider the state machines shown in figure 8.6, where A may start in initial state $A0_0$ or $A0_1$, and B starts in initial state $B0$.

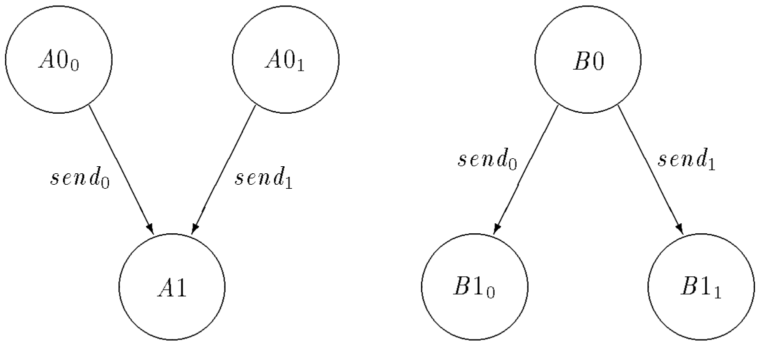


Figure 8.6: Primitive value passing

Depending on the initial state of A, the possible events are $send_0$ or $send_1$ which lead to a final state for B of $B1_0$ or $B1_1$ respectively.

We can think of events $send_0$ and $send_1$ as conveying state information from machine A to machine B. This idea is the basis of the value passing calculus.

Given the following schema text:

$$\begin{array}{|l}
X : \mathbb{P}\mathbb{N} \\
x : X \\
A0, B1 : X \rightarrowtail\!\!\!\!\!\twoheadrightarrow STATE \\
send : X \rightarrowtail\!\!\!\!\!\twoheadrightarrow EVENT \\
\hline
X = \{0, 1\} \\
A0 = \{0 \mapsto A0_0, 1 \mapsto A0_1\} \\
B1 = \{0 \mapsto B1_0, 1 \mapsto B1_1\} \\
send = \{0 \mapsto send_0, 1 \mapsto send_1\}
\end{array}$$

We will be able to show the simple example given in figure 8.6 as in figure 8.7. In the above schema, $A0$ and $B1$ are declared as (partial) injections because each element in their domain must map to a different state. Similarly *send* is declared as a (partial) injection because each element in its domain must map to a different event. Such partial injections identify, by their range, a family of states or events, and we will refer to, for example, "the family of states $A0$" or "the family of states $A0(x)$" (where x is an arbitrary member of the domain of $A0$). In future, we will not name individual states or events from such families but will refer to them by means of the functions that map onto them (eg, we refer to $A0(0)$ rather than to an actual state name such as $A0_0$).

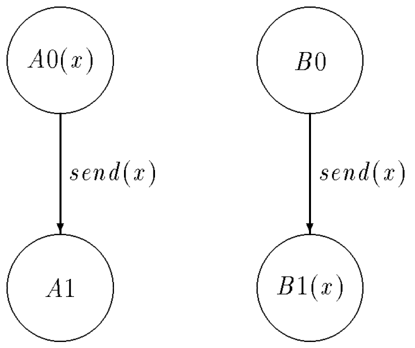


Figure 8.7: Parameterised value passing

8.6.4 Mutual Exclusion with fairness

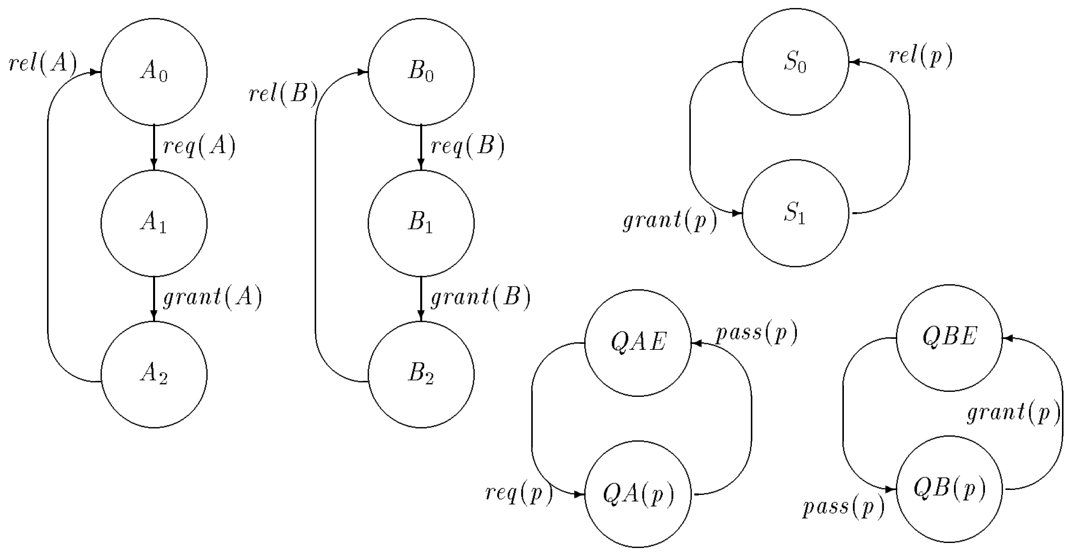
Processes A and B submit requests to enter their critical regions by adding an identity token to a two place queue modelled by QA and QB. The requests are granted when the tokens are removed from the other end of the queue. Entry to the critical region is governed by a semaphore S. When a process is in its critical region the state of S records this and also records the identity of the process.

The event calculus diagram for the model is given in figure 8.8. Some event identifiers $(req(p))$ represent a family of events and some state identifiers $(QA(p))$ represent a family of states. Since the domains of these injections contain two members, these families each contain two members as well. The diagram includes basic declarations of the functions *req*, QA etc. In an actual formal specification these declarations would be in schema form with additional predicates defining, for example, the exact domains of such functions. In this chapter such details have been omitted.

We could write out a trace of events from a the model as follows:

Event	State
	$\{A_0, B_0, S_0, QAE, QBE\}$
$req(A)$	$\{A_1, B_0, S_0, QA(A), QBE\}$
$pass(A)$	$\{A_1, B_0, S_0, QAE, QB(A)\}$
$req(B)$	$\{A_1, B_1, S_0, QA(B), QB(A)\}$
$grant(A)$	$\{A_2, B_1, S_1(A), QA(B), QBE\}$
$pass(B)$	$\{A_2, B_1, S_1(A), QAE, QB(B)\}$
$rel(A)$	$\{A_0, B_1, S_0, QAE, QB(B)\}$
$req(A)$	$\{A_1, B_1, S_0, QA(A), QB(B)\}$
$grant(B)$	$\{A_1, B_2, S_1(B), QA(A), QBE\}$

This model is considered "fair" where the model show in figure 8.3 is "without fairness". If we were to take the same disposition as we do in section 8.6.1, ie, if we were to prefer machine A over



Declarations:
$S_1, QA, QB : MACHINE \rightarrowtail\!\!\!\!\!\rightarrow STATE$
$req, grant, rel, pass : MACHINE \rightarrowtail\!\!\!\!\!\rightarrow EVENT$

Figure 8.8: Mutual exclusion with fairness

machine B, we would have different results. When machine A enters into state A_2, we are in a position of being able to fire the $req(B)$ event (as shown in the above trace). Thus machine B is now able to enter into state B_2 (after machine A releases the semaphore with a $rel(A)$ event, as shown in the above trace). Machine B is now sure to get a "turn" in its critical region before machine A gets its next trun.

8.7 A GCD algorithm, modelling parameter passing and procedure call

The greatest common divisor function may be defined recursively as follows:

$$gcd : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$\begin{aligned} &\forall\, x, y : \mathbb{N} \bullet \\ &\qquad gcd(x, 0) = x \\ &\qquad gcd(x, y) = gcd(y, x \bmod y) \end{aligned}$$

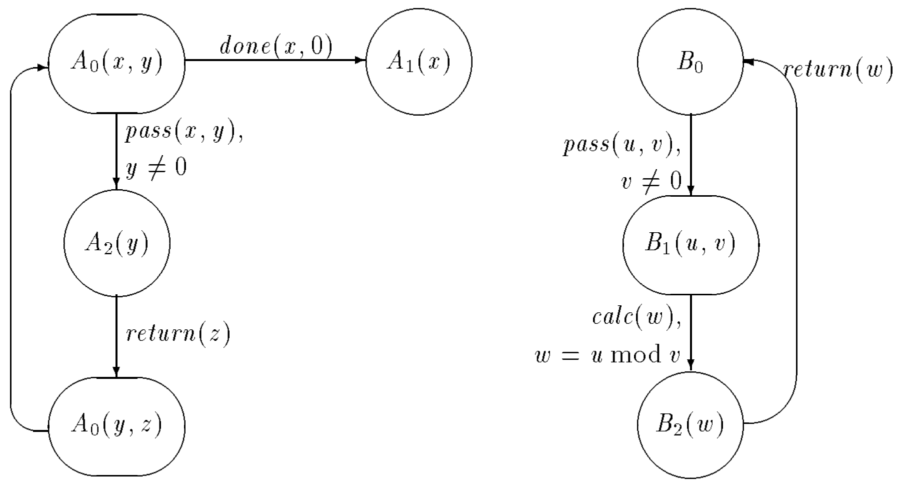
Figure 8.9 provides a composite state machine for calculating a greatest common divisor. Machine A models a "main program" and machine B models a procedure for calculating $x \bmod y$.

To see how function application is being modelled, consider the family of transitions:

$$B_1(u, v) \xrightarrow{calc(w)} B_2(w) \quad (w = u \bmod v)$$

Each different (u, v) pair is associated with a different state in the family of states $B_1(u, v)$. With each of these states we associate an event $calc(w)$ where $w = u \bmod v$. For each w the occurrence of event $calc(w)$ will take machine B into state $B_2(w)$.

In drawing machine A we have included an unnamed transition from $A_0(y, z)$ to $A_0(x, y)$. We think of this as an instantaneous unsynchronised transition in which nothing changes except the parameter



Declarations:

$A_0, B_1 : \mathbb{N} \times \mathbb{N} \rightarrowtail\!\!\!\!\!\!\!\rightarrow STATE$
$A_1, A_2, B_2 : \mathbb{N} \rightarrowtail\!\!\!\!\!\!\!\rightarrow STATE$
$done,\ pass : \mathbb{N} \times \mathbb{N} \rightarrowtail\!\!\!\!\!\!\!\rightarrow EVENT$
$calc,\ return : \mathbb{N} \rightarrowtail\!\!\!\!\!\!\!\rightarrow EVENT$

Figure 8.9: GCD algorithm, with subroutine call

names. The old value of y becomes the new value of x, and the old value of z becomes the new value of y.

8.8 Variables and Scopes

Consider machine A as given in figure 8.9. We appear to be able to follow the state history of variables x and y as the "program" progresses. Although this intuition is correct, is should be underpinned with an appreciation of the underlying formalism. Formally the diagram for machine A declares the following families of labelled transitions.

$$A_0(x, y) \xrightarrow{done(x, 0)} A_1(x) \tag{8.5}$$

$$A_0(x, y) \xrightarrow{pass(x, y)} A_2(y) \tag{8.6}$$

$$A_2(y) \xrightarrow{return(z)} A_0(y, z) \tag{8.7}$$

Consider transitions 8.6 and 8.7 of these. The identifiers x and y used in 8.6 are bound variables local to this family of transitions. The y in 8.7 is a separate bound variable associated with a different expression. As with all bound variables, the choice of identifier names is arbitrary and formally we could replace the y in 8.7 with any identifier name except x (which is already spoken for within the same scope).

However, the diagram notation we use limits our arbitrary choice of identifier name in a way that supports the intuitive understanding we have of persisting identifier values. According to this understanding, the y in 8.6 can be thought of as the same y as in 8.7 since in any trace they will take the same value when an event from 8.6 is followed by an event from 8.7.

8.9 Time

We measure time in the Event Calculus in terms of clock ticks. A clock tick is a kind of universal cosmic heartbeat, which differs from an event both in its universality and in being free of any synchronisation requirements. States may be associated with a clock function, which records the number of ticks that have occurred since that state came into existence.

We will place timing constraints on states by means of two partial functions which give the minimum and maximum times required for given events to occur after entering a given state. These will be partial functions because not all states and events will have timing constraints.

We also introduce the set *timed* which is the set of all time constrained $STATE \times EVENT$ pairs and the set *timed_states*, which is the set of states which have any time constrained events.

$$\begin{array}{|l}
minreq, maxreq : STATE \times EVENT \rightarrow \mathbb{N} \\
timed : STATE \times EVENT \\
timed\_states : \mathbb{P}\, STATE \\
\hline
timed = \mathrm{dom}\ minreq = \mathrm{dom}\ maxreq \\
\quad \wedge \\
timed\_states = \{ s : STATE \mid \exists\, e : EVENT \bullet (s, e) \in timed \} \\
\quad \wedge \\
\forall\, s : STATE;\ e : EVENT \bullet ((s, e) \in timed \Rightarrow minreq\ s \leq maxreq\ s) \\
\quad \wedge \\
\forall\, s : STATE;\ e : EVENT \bullet (s, e) \in timed \Rightarrow \neg\, (s \overset{e}{\not\leftrightarrow} s)
\end{array}$$

In the next section we will formally introduce a clock function as part of the system state. States which are subject to timing constraint have an associated clock which records how long they have been in existence, any new time constrained state that results from the event has its clock initialised to zero. Note that the final predicate of the above schema is to disallow null transitions from being time constrained. This is to avoid having to decide whether a null transition should reset a state clock.

Since timing constraints place additional restrictions on whether events can occur, we will henceforth distinguish "enabled events" (those capable of occurring if timing restraints are not considered) from "firable events" (those capable of occurring given that timing constraints are to be taken into consideration). All firable events are necessarily enabled, but not all enabled events need be firable.

In defining what we mean by a firable event we make use of the functions *minreq* and *maxreq* as follows. A time constrained state s must exist for at least $minreq(s, e)$ ticks before event e can occur. After s has persisted for $minreq(s, e)$ ticks the event e may occur if it is firable, but another clock tick (or another enabled event) may occur instead. After s has persisted for $maxreq(s, e)$ ticks or longer, the event e will occur before the next clock tick if it can. However, another tick may occur if e is not enabled. Even if e is enabled it may not occur as there may be other firable events which may occur instead.

Figure 8.10 shows a simple mutual exclusion model with time constraints on the states of machines A and B.

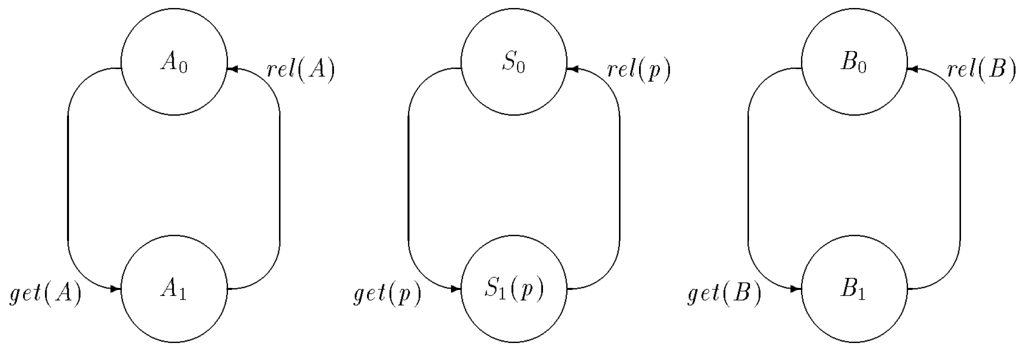
Figure 8.11 shows a partial graph of the possible behaviours of the composite machine shown in figure 8.10. In this graph we use the notation (s, n) to show that the timed state s has persisted for n clock ticks. Thus $(A_0, 2)$ indicates that state A_0 has persisted for 2 clock ticks.

Initially, although the events $get(A)$ and $get(B)$ are enabled, they are not firable because of timing constraints. After the first clock tick the event $get(B)$ becomes firable. We now have a choice of possible behaviours consisting of another clock tick or the event $get(B)$.

The trace consisting of "*tick*, *tick*", brings us to a composite state where B_0's clock has reached $maxreq(B_0, get(B))$. Some event must now occur before the next clock tick. Two events are firable, $get(A)$ and $get(B)$. If $get(B)$ now occurs we enter the composite state:

$$\{(A_0, 2), S_1(B), (B_1, 0)\}$$

State B_1 is a new member of the compound state and has its clock initialised to zero.



Declarations:

$get,\ rel : MACHINE \rightarrowtail\!\!\!\!\!\!\!\!\;\; \nrightarrow EVENT$
$S1 : MACHINE \rightarrowtail\!\!\!\!\!\!\!\!\;\; \nrightarrow STATE$

Timing Constraints:

$minreq = \{((A_0, get(A)), 2), ((A_1, rel(A)), 0), ((B_0, get(B)), 1), ((B_1, rel(B)), 1)\}$
$maxreq = \{((A_0, get(A)), 3), ((A_1, rel(A)), 1), ((B_0, get(B)), 2), ((B_1, rel(B)), 1)\}$

Figure 8.10: Timing constraints example

If, on the other hand, $get(A)$ occurs we enter the composite state:

$$\{(A_1, 0), S_1(A), (B_0, 2)\}$$

Now although B_0's clock has reached $maxreq(B_0, get(B))$ the event $get(B)$ is not enabled. At this point a tick can occur, or the event $rel(A)$ can occur.

The form of the timing constraints was chosen to allow the modelling of interrupts, time outs and to allow maximum and minimum times for complex behaviours to be calculated. We base these calculations on the maximum and minimum times required for the possible event sequences that make up the behaviour.

Consider the event sequence:

$$get(A),\ rel(A),\ get(B),\ rel(B)$$

We can deduce the minimum time this sequence can take by generating a trace which includes these events and which will always prefer an event to a clock tick:

$$tick,\ tick,\ get(A),\ rel(A),\ get(B),\ tick,\ rel(B)$$

and we can deduce the maximum time by generating a trace which always prefers a tick to an event:

$$tick,\ tick,\ get(A),\ tick,\ rel(A),\ get(B),\ tick,\ rel(B)$$

A formal specification of the timing rules for the system requires a notion of current system state, which is given in the following section.

8.10 The Dynamic model

Until now our formalisation of the Event Calculus has used a static model, in the sense that our next state functions have held information from which all possible behaviours of an Event Calculus model could be deduced.

$$\begin{tikzcd}
 & & \{\ (A_0,0), S_0, (B_0,0)\ \} \arrow[d, "tick"] & \\
 & & \{\ (A_0,1), S_0, (B_0,1)\ \} \arrow[dl, "tick"'] \arrow[dr, "get(B)"] & \\
 & \{\ (A_0,2), S_0, (B_0,2)\ \} \arrow[dl, "get(A)"'] \arrow[dr, "get(B)"] & & \{\ (A_0,1), S_1(B), (B_1,0)\ \} \\
\{\ (A_1,0), S_1(A), (B_0,2)\ \} \arrow[d, "tick"'] & & \{\ (A_0,2), S_1(B), (B_1,0)\ \} & \\
\{\ (A_1,1), S_1(A), (B_0,3)\ \} & & &
\end{tikzcd}$$

Figure 8.11: Partial graph of behaviours for the machine of figure 8.10

We now supplement this static description with a dynamic model which has a notion of the current system state. As part of this model we introduce schemas to describe how the current state is updated by the occurrence of an event or a clock tick.

The "data base schema" for the dynamic model records the composite current state and the clock value of each time constrained state within this composite state.

MachineState

$machines : \mathbb{P}\, MACHINE$
$current\_state : validstateset$
$clock : (timed\_states \cap current\_state) \nrightarrow \mathbb{N}$
$firable : EVENT \rightarrow \mathbb{B}$

$machine\ (\!|\ current\_state\ |\!) = machines$

$\forall\, e : EVENT \bullet$
$\quad firable(e) = enabled(current\_state, e)$
$\quad \wedge$
$\quad \forall\, s : current\_state \bullet$
$\qquad (s,e) \in timed \Rightarrow clock\ s \geq minreq(s,e)$

In this schema we declare the set of machines to be modelled, a set of states which gives the current state of each of these machines and a clock function to give the clock value associated with each time constrained state. We also declare a boolean function which tells us whether a given event may occur (is firable) in the given system state. An event may fire if it is enabled and if all time constrained states that will change as a result of the event have clocks that are past their minimum tick values.

We next describe the changes caused by firing an event.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{FireEvent}} \\
\hline
\Delta \textit{MachineState} \\
e? : \textit{EVENT} \\
\textit{result}! : \{\text{“}\texttt{ok}\text{”}, \text{“}\texttt{cannot fire}\text{”}\} \\
\hline
\textit{firable}\ e? \\
\quad \wedge\ \textit{current\_state}' = (\chi\, \textit{machines})(\textit{current\_state}, e?) \\
\quad \wedge\ \textit{clock}' = ((\textit{current\_state}) \ndres \textit{clock}) \cup \\
\qquad \{s : \textit{timed\_states} \mid s \in \textit{current\_state}' \setminus \textit{current\_state} \bullet s \mapsto 0\} \\
\quad \wedge\ \textit{result}! = \text{“}\texttt{ok}\text{”} \\
\vee \\
\quad \neg\ \textit{firable}\ e? \\
\quad \wedge\ \textit{current\_state}' = \textit{current\_state} \\
\quad \wedge\ \textit{clock}' = \textit{clock} \\
\quad \wedge\ \textit{result}! = \text{“}\texttt{cannot fire}\text{”} \\
\hline
\end{array}
$$

If an event which can fire is input, the new composite state is found by applying the composite next state function χ. Any new time constrained state that results from the event has its clock initialised to zero and the message “ok” is output.

If an event is input which cannot fire the system state does not change and the message “cannot fire” is output.

The final fundamental aspect of our dynamic model is the clock tick. This increments the clock value associated with each time constrained state in the current state space. However, there are some circumstances in which a tick cannot occur. Suppose we have one or more state event pairs (s, e) such that e is a firable event and $\textit{clock}\ s \geq \textit{maxreq}(s, e)$. Then an event must occur before the next clock tick.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Tick}} \\
\hline
\Delta \textit{MachineState} \\
\textit{result}! : \{\text{“}\texttt{ok}\text{”}, \text{“}\texttt{an event must occur before the next tick}\text{”}\} \\
\hline
\textit{current\_state}' = \textit{current\_state} \\
\quad \wedge\ (\neg\ \exists\, e : \textit{EVENT} \bullet (\textit{firable}\ e\ \wedge \\
\qquad \exists\, s : \operatorname{dom} \textit{clock} \bullet \textit{clock}\ s \geq \textit{maxreq}(s, e))\ \wedge \\
\quad \textit{clock}' = \{s : \textit{STATE}, n : \mathbb{N} \mid s \mapsto n \in \textit{clock} \bullet s \mapsto n + 1\} \\
\quad \wedge\ \textit{result}! = \text{“}\texttt{ok}\text{”}) \\
\vee \\
\quad (\exists\, e : \textit{EVENT} \bullet \textit{firable}\ e\ \wedge \\
\qquad \exists\, s : \operatorname{dom} \textit{clock} \bullet \textit{clock}\ s \geq \textit{maxreq}(s, e) \\
\quad \wedge\ \textit{clock}' = \textit{clock} \\
\quad \wedge\ \textit{result}! = \text{“}\texttt{an event must occur before the next tick}\text{”}) \\
\hline
\end{array}
$$

8.11 Combining the Event Calculus with Z schema calculus

Data base operations are typically described in Z by first giving a data base schema:

$$S_0 \mathrel{\widehat{=}} [\, D_0 \mid P_0 \,]$$

then defining operations on this data base with schemas having a general form:

$$S_1 \mathrel{\widehat{=}} [\, \Delta S_0;\ D_1 \mid P_1 \,]$$

The declaration D_0 will give the data structures of the data base. The predicate P_0 will describe data base invariants. D_1 will declare Input/Output identifiers associated with the update. P_1 will give any restrictions and preconditions on the form of these identifiers and will also describe how the new system state D_0' is related to the previous system state D_0.

Suppose that in an event calculus model the state of this data base is maintained by a machine M. We will provide an interpretation for a "Schema Transition" of the form:

$$M_0(S_0) \xrightarrow{S_1} M_1(S_0')$$

such that the Schema Transition describes, in terms of the event calculus, the data base update described by the schema S_1 in Z.

In formulating this interpretation we face certain difficulties. Firstly, in the Event Calculus, parameterised events accept tuples as arguments and identify arguments by their position within the tuple. In Z schemas, on the other hand, declarations do not have any particular order and arguments are identified by name. To handle this problem, we extend Z with an alphabetic ordering symbol α so that if D is a declaration, αD will be the same declaration with its components written in alphabetic order by identifier name. Thus if X and Y are basic types and:

$$D \text{ is } x, z : X;\ y : Y$$

then

$$\alpha D \text{ is } x : X;\ y : Y;\ z : X$$

In addition, we use ΘD to represent the type obtained from the declaration D by taking the cartesian product of the types of its identifiers in the order in which they are written, following Spivey (1989) we use θD to represent the characteristic tuple formed by writing out the identifiers of D. For example

$$\Theta(\alpha D) \text{ is } X \times Y \times X$$

and

$$\theta(\alpha D) \text{ is } (x, y, z)$$

These notations will allow us to construct certain tuples required in our event calculus model.

There is also a problem with respect to identifier scope. In a parameterised labelled transition, identifiers are bound variables the scope of which is the labelled transition together with any qualifying predicate. Thus

$$A_0(x) \xrightarrow{step(y)} A_1(z), \quad (y < x \wedge z = x - y)$$

could equally well be written as:

$$A_0(a) \xrightarrow{step(b)} A_1(c), \quad (b < a \wedge c = a - b)$$

In a schema, on the other hand, any identifiers declared in the schema have a scope which lasts till the end of the schema but may subsequently be reintroduced into the formal discussion by quoting the schema name. To overcome this discrepancy we use universal schema quantification. If D is a declaration restricted by a property P and if S is a schema, the declaration part of which includes all the identifiers of D such that S can be expressed in the form:

$$S \mathrel{\widehat=} [\, D;\ D_s \mid P_s \,]$$

then

$$\forall\, D \mid P \bullet S$$

represents the schema

$$[\, D_s \mid (\forall\, D \mid P \bullet P_s) \,]$$

In our usage of this notation, D and P will be the declarations and predicate of the schema describing the data base update. D_s and P_s are the additional declarations and predicates required to describe the labelled transition which will perform the data base update in the event calculus model.

We need some final conventions to generate the declarations of the parameterised states and the parameterised event needed in the event calculus model. We derive their names from those used in the schema transition

$$M_0(S_0) \xrightarrow{S_1} M_1(S_0')$$

We take M_0 and M_1 as the names of our parameterised state functions. We obtain the parameterised event function name by converting the first character of the data base update schema S_1 to lower case. We represent this name informally as s_1. This usage is informal because it does not show the derivation of the name s_1 from the name S_1. For example, if the name of the data base update schema is *Book*, the corresponding parameterised event name is *book*.

We are now ready to give the event calculus interpretation of the schema transition:

$$M_0(S_0) \xrightarrow{S_1} M_1(S_0')$$

where S_0 is a data base description schema

$$S_0 \mathrel{\widehat{=}} [\, D_0 \mid P_0 \,]$$

and S_1, which describes a data base update operation, has the form:

$$S_1 \mathrel{\widehat{=}} [\, \Delta S_0, D_1 \mid P_1 \,]$$

Our interpretation of this text at the event calculus level will be:

$$\forall\, D_0;\ D_0';\ D_1 \bullet S$$

where

S

S_1;
$M : MACHINE$
$M_0, M_1 : \Theta\alpha D_0 \mathrel{+\!\!\!\!\rightarrow} states\ M$
$s_1 : \Theta\alpha D_1 \mathrel{+\!\!\!\!\rightarrow} EVENT$

$M_0(\theta\alpha D_0) \xrightarrow{s_1(\theta\alpha D_1)} M_1(\theta\alpha D_0')$

8.12 A Distributed seat booking system

As an example of the techniques described above, we specify a simple seat booking system in which bookings can be made from a number of different nodes. We use Z schemas to describe the data base and the update and enquiry operations to be performed upon it. We use an Event Calculus diagram to describe the manner in which each node is able to gain access to the data base. Together with the rules of interpretation given above, these two parts of the system description generate a formal model, in Z, of the next state functions for the system's component state machines.

We introduce two basic types, the set of all seats and the set of names for people who may book the seats.

$[SEAT, NAME]$

The state of the data base is described in the following schema using a function from seats to names. Seats which are not in the domain of this function are still free. There is an implicit invariant which disallows double booking of a seat. This invariant arises from the declaration of the relationship between seats and names as a partial function.

$$\begin{array}{|l}
\textit{Booking} \\
\hline
booked : SEAT \nrightarrow NAME \\
free : \mathbb{P}\, SEAT \\
\hline
free = SEAT \setminus \mathrm{dom}\, booked \\
\hline
\end{array}$$

We specify an enquiry operation which will return the set of free seats.

$$\begin{array}{|l}
\textit{Enquire} \\
\hline
\Xi Booking \\
free! : \mathbb{P}\, SEAT \\
\hline
free! = free \\
\hline
\end{array}$$

We specify an operation to book a seat. The operation takes as inputs a seat and a name. It has the precondition that the given seat should not be already booked and, if this is satisfied, it adds the new booking to the data base. There will be no need to specify what happens if the seat is already booked, as the event calculus part of the specification will render this event impossible by specifying a data validation check.

$$\begin{array}{|l}
\textit{Book} \\
\hline
\Delta Booking \\
seat? : SEAT \\
name? : NAME \\
\hline
seat? \in free \\
booked' = booked \cup \{seat? \mapsto name?\} \\
\hline
\end{array}$$

We now associate these schema with an Event Calculus diagram that models a system in which multiple nodes are able to share access to the data base. Figure 8.12 shows an Event Calculus diagram which, together with the seat booking specification, provides the formal specification for the system.

The seat bookings data base is maintained by machine A. There are two possible operations which can be performed on the data base, a seat booking and an enquiry. These are denoted in the Event Calculus diagram by writing:

$$Book \vee Enquire$$

This declares the two schema transitions:

$$A_0(Booking) \xrightarrow{Book} A_0(Booking')$$

and

$$A_0(Booking) \xrightarrow{Enquire} A_0(Booking')$$

There is a possible ambiguity of notation here since the expression

$$Book \vee Enquire$$

could also be taken to be a single schema formed by the "schema or" of *Book* and *Enquire*. To avoid this, we do not allow schema expressions other than schema names to be written on an event calculus diagram.

The data base update that occurs when a seat is booked is described by the schema transition:

$$A_0(Booking) \xrightarrow{Book} A_0(Booking')$$

The meaning we give to this schema transition can be expressed as:

$$\forall D \bullet S$$

Where D is the declaration part of the *Book* schema and S is the schema:

S

$Book$
$A : MACHINE$
$A_0 : (SEAT \nrightarrow NAME) \times \mathbb{P}\, SEAT \nrightarrow states\ A$
$book : NAME \times SEAT \nrightarrow EVENT$

$$A_0(booked, free) \xrightarrow{book(name?, seat?)} A_0(booked', free')$$

Machines B, C and D are three nodes which originate booking transactions. B performs its bookings when in state B_1, C in state C_1 and so on. Since access to states B_1, C_1 and D_1 is mutually exclusive, the details of the booking operation are shown in a single machine X which runs whenever B is in state B_1 or C is in state C_1 or D is in state D_1. Our intention is for X to be thought of as the common logic shared by all booking nodes, not as a separate machine invoked by a booking node when it wishes to perform a transaction.

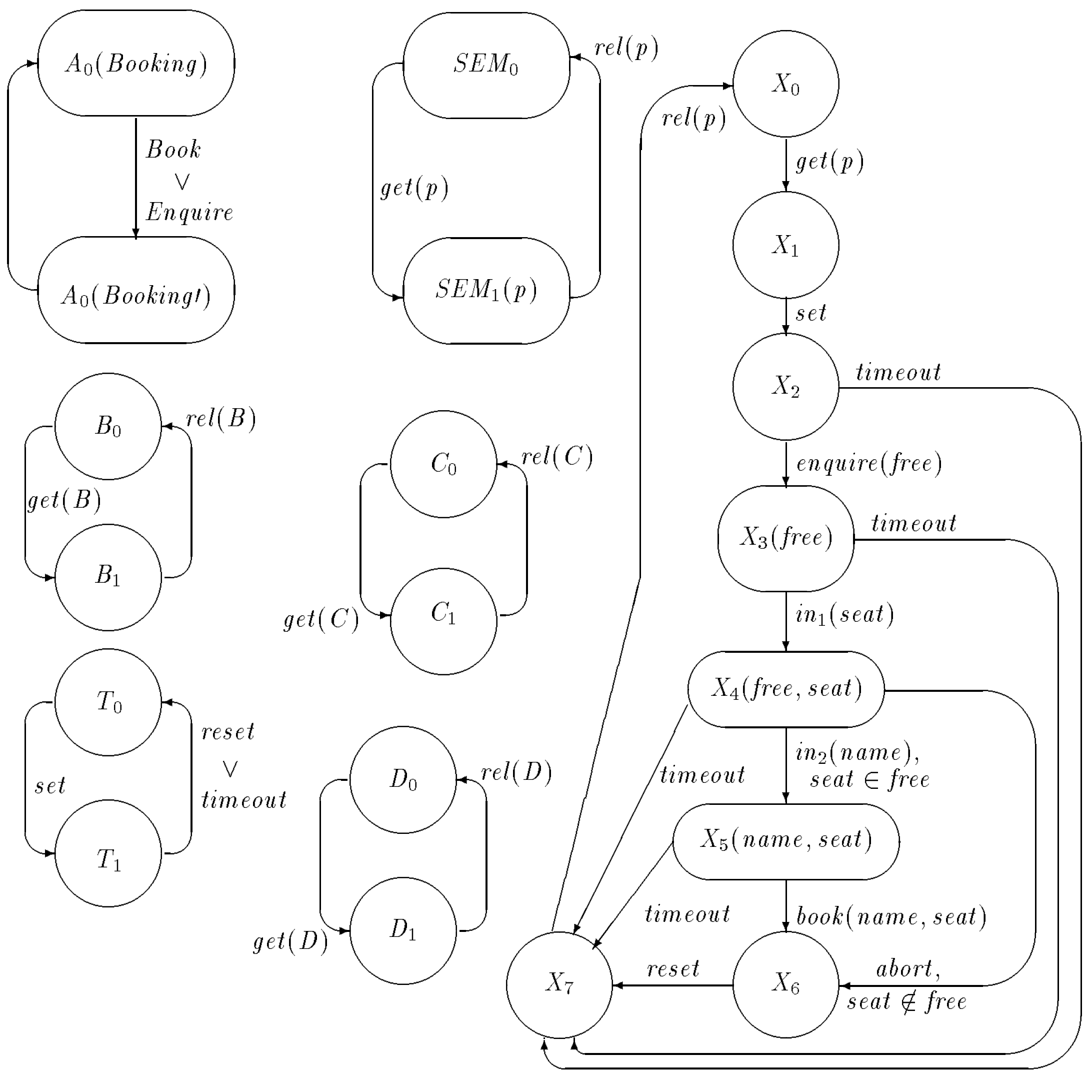
SEM is provided to control the mutually exclusive access to machine X, while T functions as a watchdog timer which will cause the booking operation to be forcibly aborted if it cannot be completed in a set time. This is achieved by placing a time constraint on the *timeout* event associated with state T_1 and on the *timeout* event associated with various states of machine X. In our specification, it is actually possible for machine X to continue with its seat booking operations when T is ready to *timeout*. However, the booking operations are not allowed to take any observable time (they must all be completed before the next clock tick).

8.13 References

Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Computer Science. London: Prentice Hall International.

Milner, R. (1989). *Communication and Concurrency.* Computer Science. London: Prentice Hall International.

Spivey, J. M. (1989). *The Z Notation: A Reference Manual.* Computer Science. London: Prentice Hall International.



Declarations:	Time Constraints:
$A_0 : (SEAT ⇸ NAME) \times \mathbb{P}\, SEAT ⤔ states\ A$	
$X_4 : \mathbb{P}\, SEAT \times SEAT ⤔ states\ X$	
$X_5 : NAME \times SEAT ⤔ states\ X$	$minreq(T_1, timeout) = 10$
$X_3 : \mathbb{P}\, SEAT ⤔ states\ X$	$maxreq(T_1, timeout) = 10$
$S_1 : MACHINE ⤔ states\ S$	$maxreq(X_2, timeout) = 0$
$enquire : \mathbb{P}\, SEAT ⤔ EVENT$	$\forall s : range\ X_3 \bullet maxreq(s, timeout) = 0$
$book : NAME \times SEAT ⤔ EVENT$	$\forall s : range\ X_4 \bullet maxreq(s, timeout) = 0$
$get,\ rel : MACHINE ⤔ EVENT$	$\forall s : range\ X_5 \bullet maxreq(s, timeout) = 0$
$in_1 : SEAT ⤔ EVENT$	
$in_2 : NAME ⤔ EVENT$	

Figure 8.12: Seat booking, with mutual exclusion & time out