An Introduction to CSP

by

J.W. Sanders

Technical Monograph PRG-65
ISBN 0-902928-47-1
March 1988

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD
England

An Introduction to CSP

J.W.Sanders

Programming Research Group
8 - 11 Keble Road, OX1 3QD, Oxford

0. Introduction

During a CSP course (as with any other) it is easy to lose sight of the wood for the trees; after spending several hours immersed in the formal properties of an operator, one is apt to forget why the operator was deemed to have been important in the first place. It has thus been found convenient to begin CSP courses with a lecture whose purpose is to provide an overview and informal introduction to the features of CSP. This complements the style of Hoare's textbook *[2]* (which develops CSP systematically and gradually) and leaves the lecturer free in the rest of the course to pursue a more rigorous presentation following the book .

In this paper we attempt to give such an introduction. Particular features are:

(a) the use of a small example to introduce the notation, and demonstration of how CSP can be used to model a potentially complex system by building the specification in small steps;

(b) early emphasis on the combinators for internal and external choice;

(c) contrast between CSP and the notation of finite automata and regular expressions (a contrast which our experience suggests is helpful to the audience); and

(d) a brief section on the development of systems emphasizing the use of laws.

Several extensions of our example are proposed as exercises. These are not primarily CSP exercises (a booklet of which is available from the publisher to accompany *[2]*) but have been included to provide practice in structuring specifications. Such exercises might form the basis of a workshop held towards the end of the course.

1. What is CSP?

CSP, a theory of Communicating Sequential Processes, attempts to provide a notation for expressing and reasoning about systems of concurrent processes. By *expressing* we mean designing, specifying and implementing: without a formal notation it is difficnlt to describe a process precisely enough to modify it or to contrast it with contending designs. By reason*ing* a*bout* we mean modifying, developing and verifying such descriptions correct: the notation must support a body of knowledge sufficient to enable all this to be done. And in *systems of concurrent processes* we iuclude multiprocessing systems, operating systems, distributed systems, systems using remote procedure calls, and systolic arrays of processors.

To meet these aims, CSP offers a succinct mathematical notation for the description of processes and a way (alphahets) of controlling the level of abstraction of these descriptions. Processes can be structured using combinators for parallel composition, external choice, internal choice, communication, abstraction and sequential composition: these combinators are introduced in section 2. The resulting notation is expressive enough to describe the range of examples mentioned in the previous paragraph, yet circumscribed sufficiently to support a body of fairly simple laws and a semantic theory. The complete semantics is too complex for general use; fortunately it is approximated by a hierarchy of simpler semantic models one of which is usually sufficient to reason about properties of the system under consideration. We say more about this hierarchy in section 3.

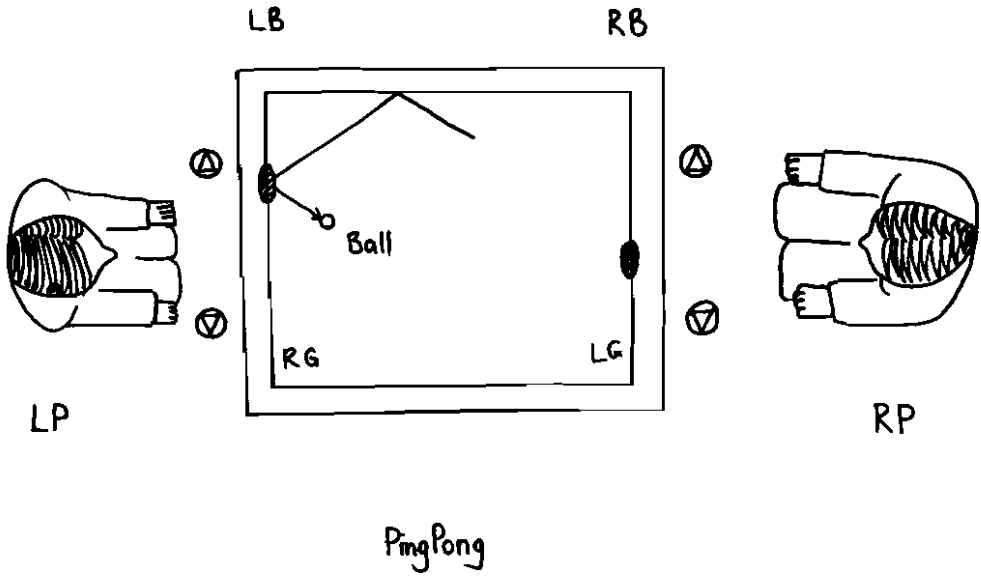
A process can be described either by CSP syntax, using the combinators mentioned in the previous paragraph, or by semantics, using the simplest model in the hierarchy which is sufficient to capture the intended behaviour. However for the purpose of executing a description in CSP, a language derived from a subset of it, called occam, has been implemented by INMOS (see *[3]*). The choice of subset has naturally beeu determined by matters of efficiency which are absent in the larger language. Consequently it is convenient to specify, reason about and refine systems in CSP and encode using occam only at the very last stage of development.

2. PingPong

This section contains the main part of the paper; in it CSP is used to model a game which can be described informally as follows.

2.0. Requirements

PingPong is a game for two players who sit at opposite ends of a small table. Set into the table is a horizontal video monitor which displays a ball bouncing around inside a four-walled court. The wall nearest the right-hand player, RP, is the goal, LG, of the left-hand player, LP, and vice versa. To defend his opponent's goal, RP controls a bat, RB, which can be moved along LG by pressing a "move right" button or a "move left" button (judged from the viewpoint of RG), or neither. The ball starts in the middle of the court at a random angle (not parallel to the goals) and collides elastically with the bats and walls. But if it reaches a player's goal, that player wins and the session ends.



Our task is to describe PingPong in CSP. The first difficulty is to decide at what level to model the game. We could describe the time at which it starts, the time it finishes, and the winner. Or we could describe, second by second, the intensity of the screen and the spatial movements of both players. Evidently the second description contains far more information than the first which is called more high-level or more abstract.

We shall endeavour to focus on an intermediate description which ignores time and most physical attributes of the components of the game. We conceive of the game as proceeding as the two players control their bats by button and the bats deflect the ball which rebounds around the court. Our first step is thus to isolate five interacting **processes:**

LP	the left-hand player	RP	the right-hand player
LB	the left-hand bat	RB	the right-hand bat
	Ball	the ball.	

It is our intention to describe the game as proceeding with the evolution in parallel of these processes

PingPong = LP || RP || LB || RB || Ball.

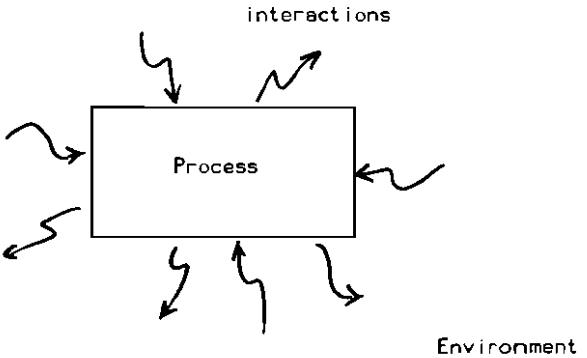
Although || denotes an infix operator with two processes as arguments and a single process as its result, we have relied on its associativity to omit parentheses. It has yet to be decided what a process is and how || should behave; but intuition dictates that placement of components in parallel ought to be associative and that we ought to be able to use || to modularise the specification as outlined above. This, after all, is the principle of top-down description, exploiting concurrency. It is similar to the use of procedures in structuring a sequential system in top-down fashion, and indeed generalises it. If our description is to be successful, || must permit the five self-contained processes to interact in just the right way!

In outline, a process is capable of performing certain actions (the left-hand player can press buttons and the left-hand bat can move up and down) and when two processes are placed in parallel (LP || LB) the components must synchronise on their common actions (the left-hand bat responds to the left-hand player's wishes) although an action which can be performed by only one of them occurs whenever that process permits it to (the bat is insensitive to its player's victory).

But let us first discuss the players, LP and RP.

2.1. Players

Think of a process as being a black box. Its internal constitution is concealed from us but we can nonetheless describe it by describing its interactions with its environment. This is a standard expedient in scientific methodology and is traditionally accompanied by the rather trivial diagram



The kind of interaction we are interested in determines the level of abstraction of the model. In PingPong we are not concerned with the left-hand player's clothing, his drinking habits, or even whether he is breathing: we simply wish to describe his inclination to press the two buttons at his disposal. The relevant **events** in which LP can engage are

left	push the left-hand button
right	push the right-hand button
stay	push neither button.

We say that these events constitute the **alphabet** of LP,

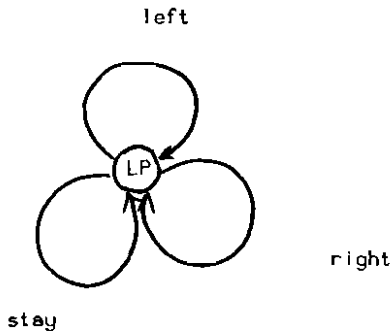
$$\alpha(LP) \triangleq \{left, right, stay\}.$$

Had we chosen to model the left-hand player's exclamations during the game then we would have had to incorporate his vocabulary into this alphabet; had we elected not to model his conscious decision to move the bat neither left nor right, we would have deleted stay from it.

The left-hand player can push buttons as frequently as he likes and in any order. The choice of what he does is determined by factors not revealed at this level of abstraction. So in our model LP can, at any stage, engage in any of the events in its alphabet.

Exercise. Describe LP using (a) the diagram of a finite automaton;
(b) a grammar and its language.

Answer. (a)



(b) A regular grammar is

LP = left LP | right LP | stay LP | ϵ

(where ϵ denotes the empty string) and its language consists of all finite sequences of left, right and stay

{left, right, stay}*. //

The way in which we describe LP using CSP makes apparent that the choice between the different events in its alphabet is made internally by LP and cannot be influenced by its environment. Such interpretations (whether a process decides internally to perform an event or whether it does so in consultation with its environment) are absent from the notation of finite automata. In CSP we write

$$LP = (left \rightarrow LP) \sqcap (right \rightarrow LP) \sqcap (stay \rightarrow LP). \qquad (1)$$

This equation defines LP **recursively**. Some care is obviously necessary to ensure that such a definition makes sense; for instance LP is not defined by either

$$LP = LP \qquad \text{or} \qquad LP = LP \sqcap LP.$$

This point must be pursued in the body of the CSP course (see *[2]*, page 97).

The infix combinator $\sqcap$ combines two processes and yields a process which is an internal choice between them, that is, made by factors beyond the control of the environment. It is called **internal**, or **nondeterministic**, **choice** and the reader's intuition will almost certainly direct that, whatever its definition, $\sqcap$ ought to be associative, commutative and idempotent. This is seen to be true in the course (see *[2]*, chapter 3) and as usual we have made use of associativity in omitting parentheses in the right-hand side of (1).

The symbol $\rightarrow$ combines an event and a process and produces a second process capable of performing the event and subsequently behaving like the first process. This is called **prefixing** and $\rightarrow$ is pronounced **then**.

A process thus described, determines (like the automaton above) the set of finite sequences of events in which it is prepared to engage. These finite sequences (which include the empty one) are called the **traces** of the process LP and their aggregate is denoted

$$traces(LP) \qquad (= \{left, right, stay\}^*). \qquad (2)$$

We often think of (1) as the syntax for LP and (2) as its trace semantics.

So far we have ignored termination. Let us suppose that although the game need not terminate, if it does either LP wins or RP wins. Ignoring the manner in which they celebrate this victory, we choose to augment the alphabet of LP with the events

$$\{lwin, rwin\}.$$

After engaging in either of these events LP terminates. We write

$$SKIP$$

for a process which models such termination. Although it engages in no event (its only trace is the empty one), it can be followed in sequence by another process, in which case the sequential composition behaves like the second process. This means, writing $;$ for sequential combination and P for an arbitrary process, that provided SKIP and P are at the same level of abstraction (have the same alphabets),

$$SKIP ; P = P.$$

SKIP does not model deadlock (a death after which there is no life?) – this is done by another process, STOP (see [2], page 25). SKIP is more like sleeping beauty awaiting a princely successor, and its use here enables the left-hand player and his opponent to play another match.

At the end of the game the left-hand player behaves like either

$$(lwin \rightarrow SKIP) \qquad \text{or} \qquad (rwin \rightarrow SKIP).$$

The choice between these processes is not made by the left-hand player (would he ever choose the latter event?), so $\sqcap$ is an inappropriate combinator. The choice is made by LP's environment because it depends on the position of the ball. We must thus introduce a new, environment-controlled, choice combinator: it is written $[\!]$ and called **external**, or **deterministic, choice.** So at the end of the game LP behaves like

$$(lwin \rightarrow SKIP) \;[\!]\; (rwin \rightarrow SKIP)$$

and the choice between such behaviour and nontermination (as described in our previous version of LP) is again made by LP's environment - it depends on whether the ball is at a goal. Thus finally

$$\alpha(LP) \triangleq \{left, right, stay, lwin, rwin\}$$

$$\begin{aligned} LP \triangleq\ & ((left \rightarrow LP) \sqcap (right \rightarrow LP) \sqcap (stay \rightarrow LP)) \\ & [\!] \\ & ((lwin \rightarrow SKIP) \;[\!]\; (rwin \rightarrow SKIP)). \end{aligned} \tag{3}$$

After having completed the specification of LP, we next observe the similarity between LP and RP.

Exercise. Define RP in CSP. //

There is a systematic way to exploit such isomorphism, called **relabelling**. Let us introduce three new events left', right' and stay', and define a function

$$f : \alpha(LP) \rightarrow \{left', right', stay', lwin, rwin\}$$

by setting

$$f(left) = right'$$

$$f(right) = left'$$

$$f(stay) = stay'$$

$$f(lwin) = lwin$$

$$f(rwin) = rwin.$$

(Would it have made any difference to the model to write f(left) = left' etc. ? It will become clear in the second exercise of section 2.2 why we make the choice above.)

We define a process named f(LP) as follows: its alphabet is $f(\alpha(LP))$ and it engages in event f(e) iff LP engages in event e. This "lifts" the function f from events to processes, and enables us to define

$$RP \triangleq f(LP).$$

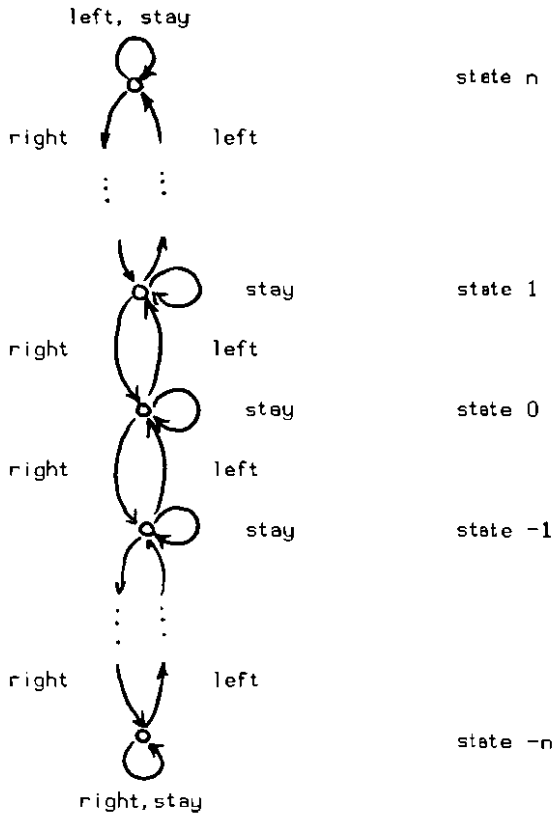
Why are three events in $\alpha(LP)$ primed and two not? We have decided that processes in parallel will synchronise on their common events. In parallel, LP and RP must be able to perform independently the events which control their own bats: hence the three primed events. However they must together engage in the win event which terminates the game: and hence lwin and rwin are common to the alphabets of LP and RP.

2.2. Bats

The left-hand bat, LB, starts in the centre of RP's goal and when it moves, does so in small uniform jumps left or right (directed from the point of view of LP) as far as the extremes of the court. We deem LB to be capable of at most n moves in either direction from its initial position, for some $n \in \mathbb{N}$ at least 1. In the implementation n will be quite large, its actual value being determined by matters of graphical and computational efficiency. But let us abstract from these and treat n as a constant of the specification.

Exercise. Describe the position of the left-hand bat using (a) a finite automaton; (b) a grammar.

Answer. (a)



(b) A grammar for this automaton is

$LB = LB_0$

$LB_t = \mathsf{left}\ LB_{t+1} \mid \mathsf{right}\ LB_{t-1} \mid \mathsf{stay}\ LB_t \mid \in$ provided $|t| \leqslant n-1$

$LB_{-n} = \mathsf{left}\ LB_{-n+1} \mid \mathsf{right}\ LB_{-n} \mid \mathsf{stay}\ LB_{-n}$

$LB_n = \mathsf{left}\ LB_n \mid \mathsf{right}\ LB_{n-1} \mid \mathsf{stay}\ LB_n.$

The language of this grammar is of course identical to that for LP. At this level of abstraction we have made no use of the "position marker" n and were we not to make use of it later in the specification, our description would be unnecessarily complicated (not **fully abstract**). //

In contrast to the specification of LP, we need (in order to keep the bat within the court) to keep track of the position of the left-hand bat. For convenience we choose, as we did with LP, to specify LB in two steps. Again the second version supersedes the first and so we use the same name, LB, in both. The first specification, following the exercise, makes no use of the state. We express LB as a process in CSP

$\alpha(LB) = \{\mathsf{left}, \mathsf{right}, \mathsf{stay}\}$

$LB = LB_0$

$LB_t = (\mathsf{left} \rightarrow LB_{t+1}) \,[]\, (\mathsf{right} \rightarrow LB_{t-1}) \,[]\, (\mathsf{stay} \rightarrow LB_t)$
provided $-n \leqslant t-1 \wedge t+1 \leqslant n$, that is, $|t| \leqslant n-1$

$LB_{-n} = (\mathsf{left} \rightarrow LB_{-n+1}) \,[]\, (\mathsf{right} \rightarrow LB_{-n}) \,[]\, (\mathsf{stay} \rightarrow LB_{-n})$

$LB_n = (\mathsf{left} \rightarrow LB_n) \,[]\, (\mathsf{right} \rightarrow LB_{n-1}) \,[]\, (\mathsf{stay} \rightarrow LB_n).$ (4)

Equations (4) use **mutual recursion** to define LB in terms of processes LB_t (where $t \leqslant n$ records the position or "state" of the bat). It is important, and this distinction does not appear in the finite automata for LB and LP, that the choice between events left, right and stay is not made by the bat – it is an external choice offered to the environment of LB. That is why we have used the combinator $[]$, representing external choice. CSP provides an alternative notation for such processes, constructed as the external choice of (sub) processes which offer distinct choices of first event to their environment. As an example of this alternative notation, LB_t can be

expressed

$$LB_t = (left \rightarrow LB_{t+1} \mid right \rightarrow LB_{t-1} \mid stay \rightarrow LB_t).$$

Here $\mid$, pronounced **choice**, is a compressed form of $\Box$ and represents an external choice between distinct events. It is used together with a menu of these distinct events (in this example, left, right and stay), whilst $\Box$ is a combinator of processes.

We have permitted LB_n to engage in the event left, but without changing its state (that is, its position). The alternative would have been to bar it from performing a left by instead defining

$$LB_n = (right \rightarrow LB_{n-1}) \Box (stay \rightarrow LB_n). \qquad (5)$$

We shall see shortly why this yields a mismatch between LP and LB which can ultimately lead to deadlock. But first let us interrupt the design of LB to give the reader an opportunity for revision.

Exercise. Define RB.

Answer. If $f' = f\restriction\{left, right, stay\}$, where f was defined in section 2.1 (here $f\restriction E$ denotes the restriction of function f to set E), then

$$RB = f'(LB).$$

This time it is important that LB and RB be mirror images of each other in order for RB to present the correct choice of events to its environment in its extreme states. //

Before making use of the state t of LB_t we need to discuss parallel combination in more detail. LP $\parallel$ LB represents the parallel combination of LP and LB: it is the process which results from LP and LB interacting in parallel. Its alphabet is

$$\alpha(LP \parallel LB) \triangleq \{left, right, stay, lwin, rwin\} \quad (= \alpha(LP) \cup \alpha(LB))$$

and an event common to both processes is performed by LP $\parallel$ LB when and only when it is performed by both LP and LB. So, from the start, LP chooses internally between left, right and stay, and LB allows it to make this choice and so engages in the same event. It is, after all, the player who controls the next position of the bat,

and not the bat which determines the next action of the player. The processes proceed in step, synchronising on their common events.

Had we made the modification to LB suggested by equation (5), LP || LB would proceed in the same way until LB reached an extreme state - say LB_n. Now if LP insisted on engaging in the event `left` then since LB also contains `left` in its alphabet but is not prepared to engage in it at this stage, deadlock occurs. On the other hand `lwin` belongs to the alphabet of LP and so it can occur whenever LP is prepared for it to do so. This does not lead to deadlock because `lwin` is not in the alphabet of LB, and so LB does not synchronise with it.

In summary, a common event in P||Q occurs when and only when it occurs in both P and Q, and an event in the alphabet of exactly one of P and Q occurs when that process allows it to, in which case the other process takes no notice, makes no progress, and in short remains unchanged.

Since we are defining the five processes LP, RP, LB, RB and `Ball` to he self-contained, some synchronisation is necessary between them. When the ball reaches the left-hand goal, for example, it must find out the position of the defending bat so that it can determine whether the game ends or whether the ball should rebound. This implies, from the meaning just ascribed to ||, that LB must always be prepared to offer its position (containing the ball) to the environment : an offer which will be accepted by the ball whenever it reaches the left-hand goal. For $|t| \leq n$ we must thus incorporate in the alphabet of LB the events (to be made common with those of `Ball`)

$$lposition_t.$$

For reasons which will become apparent in section 2.3, we rewrite these events

`lposition!t.`

The modification to LB is thus to augment its alphabet by the $2n+1$ new events

$$\{lposition!t : |t| \leq n\}$$

and to permit its environment to read them :

$$LB = LB_0$$

$$\begin{aligned} LB_t = \; & (left \rightarrow LB_{t+1} \\ & |right \rightarrow LB_{t-1} \\ & |stay \rightarrow LB_t \\ & ||position!t \rightarrow LB_t) \end{aligned} \qquad \text{provided } |t| \leq n-1$$

$$\begin{aligned} LB_{-n} = \; & (left \rightarrow LB_{-n+1} \\ & |right \rightarrow LB_{-n} \\ & |stay \rightarrow LB_{-n} \\ & ||position!t \rightarrow LB_{-n}) \end{aligned}$$

$$\begin{aligned} LB_n = \; & (left \rightarrow LB_n \\ & |right \rightarrow LB_{n-1} \\ & |stay \rightarrow LB_n \\ & ||position!t \rightarrow LB_n). \end{aligned} \qquad (6)$$

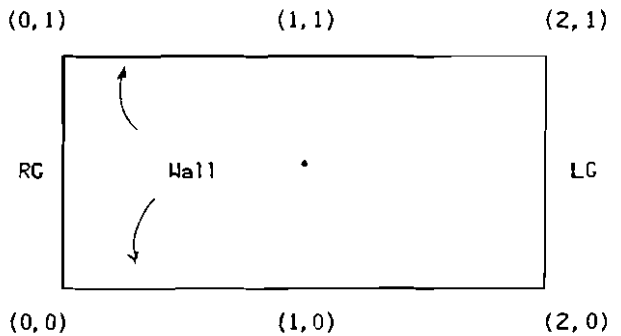
What happens when the ball reaches the goal and the bat, only one jump away, is frantically directed by its player to cover the ball's progress? Does the bat respond to its player's wish or does it reply to the ball's request for its position? The answer is simple: it performs whichever event reaches it first by virtue of preceding its opponent in the trace of events from LB's environment.

Exercise. Define RB. //

2.3. Ball

It is now convenient to settle upon some notation for the court. The set of possible positions for the ball is

$$Court \triangleq \{(x, y) \in R^2 \mid 0 \leq x \leq 2 \wedge 0 \leq y \leq 1\} = [0, 2] \times [0, 1]$$



The segments of the boundary of the court which interest us are:

$Wall$	$\triangleq \{(x, y) \in R^2 \mid 0 < x < 2 \wedge y \in \{0, 1\}\}$	top and bottom walls;
RG	$\triangleq \{(x, y) \in R^2 \mid x = 0 \wedge 0 \leq y \leq 1\}$	RP's goal;
LG	$\triangleq \{(x, y) \in R^2 \mid x = 2 \wedge 0 \leq y \leq 1\}$	LP's goal;
$Court^0$	$\triangleq Court - (Wall \cup LG \cup RG)$	interior of the court.

These definitions have involved a quick executive decision: are the corners $(0, 0)$, $(0, 1)$, $(2, 1)$ and $(2, 0)$ to be included in the wall or the goals? Resorting to the informal description does not resolve this point, and this is typical of realistic requirements documents which ignore such seemingly pedantic issues. In practice, since these points must be resolved in a formal specification, the user should be consulted. Here we assume that the user has given us freedom to do what is easiest and have decided to make the corners part of the goal: otherwise we would have had to describe the ball's reflection from them.

In this notation, LB starts at $(0, 1/2)$ and is at $(0, 1)$ when it is in its left-most state. Of course we could have used the position of LB as the subscript in definition (6), but the one given there seems simpler - even in the context of the extensions suggested in section 2.4.

The ball is to start at $(1, 1/2)$ in a random direction and bounce elastically (so that angle of incidence equals angle of reflection) from the walls and bats. Just as the bats move in small jumps, so does the ball. Its subsequent position depends on its present one, its direction, and (sometimes) on the position of the bats. Thus the state of the ball is taken to he a pair of vectors

(p, d) where p is the present position of the ball, $p \in \mathrm{Court}$, and
d is the direction of the ball, $d \in R^2$, $|d|=1$.

For convenience we write $p = (p_x, p_y)$ and $d = (d_x, d_y)$. We are assuming that the speed of the ball is constant, that there is no friction, that there is no spin imparted to the ball by the bats, and so on.

Since the ball starts in a random nonvertical direction,

$$\mathrm{Ball}_1 \in \prod_{d_x \neq 0} \mathrm{Ball}((1, 1/2), d)$$

where is a $\prod$ prefix form of internal choice, used in exactly the same way that Σ is used as a prefix form of the infix symbol $+$. For convenience we have ignored the discrete nature of d_x : it would qualify the predicate $d_x \neq 0$. So too would any condition we cared to impose to stop the ball bouncing nearly vertically from the outset.

Away from the edges of the court the ball moves in the same direction, from state (p, d) to state $(p+d, d)$. It penetrates a goal unless deflected by a bat in which case its state changes from (p, d) to $((p_x-d_x, p_y+d_y), (-d_x, d_y))$. Similarly at the wall the ball changes from state (p, d) to state $((p_x+d_x, p_y-d_y), (d_x, -d_y))$: this occurs whenever the ball lies in Court^0 but its subsequent position does not.

Thus to define the ball's movement we suppose that it is in state $\mathrm{Ball}(p, d)$, with $p \in \mathrm{Court}^0$, and consider the cases:

if $p+d \in Court^0$ then the ball moves to state $(p+d, d)$;

if $p+d$ lies on or over the wall then the ball moves to state
$((p_x+d_x, p_y-d_y), (d_x, -d_y))$;

if $p+d$ lies on or past a bat then the ball moves to state
$((p_x-d_x, p_y+d_y), (-d_x, d_y))$;

if $p+d$ lies on or past the left-hand goal not covered by a bat then lwin occurs and the ball returns to the middle of the court;

if $p+d$ lies on or past the left-hand goal not covered by a bat then rwin occurs and the ball returns to the middle of the court.

(7)

The CSP syntax for the conditional construct (**if** B **then** P **else** Q) is

$P \lhd B \rhd Q$.

(The case for such an infix notation is made in *[1]*.) Thus

if B1 **then** P1 **else if** B2 **then** P2 **else** P3

becomes

$P1 \lhd B1 \rhd (P2 \lhd B2 \rhd P3)$

or, in two-dimensional form which is easier to read and write in the case of larger texts,

$$\begin{array}{l} \quad P1 \\ \lhd\ B1\ \rhd \\ \quad P2 \\ \lhd\ B2\ \rhd \\ \quad P3\ . \end{array}$$

Does our omission of brackets here imply that the ternary combinator $\_ \lhd \_ \rhd \_$ is associative in its extremities? Certainly not: in general

$$P1 \lhd B1 \rhd (P2 \lhd B2 \rhd P3) \neq (P1 \lhd B1 \rhd P2) \lhd B2 \rhd P3,$$

although associativity does hold in the simple case B1 = B2. However in this document, unless written otherwise, we shall bracket conditionals from the right, and these correspond with a top-to-bottom reading of the predicates Bi. For example

$$P1 \lhd B1 \rhd (P2 \lhd B2 \rhd (P3 \lhd B3 \rhd P4))$$

is more legibly displayed

$$\begin{array}{l} \quad P1 \\ \lhd\ B1\ \rhd \\ \quad P2 \\ \lhd\ B2\ \rhd \\ \quad P3 \\ \lhd\ B3\ \rhd \\ \quad P4\ . \end{array}$$

Now we can re-write the ball's change of state expressed by the conditionals (7)

$$\begin{array}{lll} Ball(p, d) \triangleq & (& Ball(p+d, d) \\ & \lhd & p+d \in Court^0 \rhd \\ & & Ball((p_x+d_x, p_y-d_y), (d_x, -d_y)) \\ & \lhd & p+d \text{ on/over } Wall \rhd \\ & & Ball((p_x-d_x, p_y+d_y), (-d_x, d_y)) \\ & \lhd & p+d \text{ on/past a bat} \rhd \\ & & lwin \rightarrow Ball_1 \\ & \lhd & p+d \text{ on/past LG without bat} \rhd \\ & & rwin \rightarrow Ball_1 \quad) . \end{array} \tag{8}$$

We must next ensure that the description of Ball is self-contained, by incorporating in it the communications with its environment (the bats) which determine whether p+d is on or past the position of a bat. Since, for example, LB determines which of the events lposition!j occurs, Ball must be prepared for any of them. Ball thus offers an external choice between all events in the set

$$\{lposition!j : |j| \leq n\}$$

and its subsequent behaviour depends on which j is communicated. To express this in the notation introduced so far involves a daunting proliferation of cases. Fortunately CSP offers abbreviated notation, by letting the value j be a variable x and defining subsequent behaviour in terms of x. We thus define the complementary half of a communication event (of which lposition!j is the **send**, or output, half) to be the **receive** (or input) event

lposition?x

where x is instantiated to whatever value matches the complementary send event. Here ! stands for **output**, ? for **input**, and the two, like particle and antiparticle, combine to enable the resulting process to progress with a net communication from the outputting sub-process to the inputting one.

It remains to modify Ball to read the position of the left-hand bat LB when the ball is in the right-hand goal RG, and symmetrically. Of course we must convert the state being output by LB

$$x \in \{-n, -n+1, \ldots, -1, 0, 1, \ldots, n-1, n\}$$

to the corresponding value

$$x' \triangleq (x+n)/(2n) \in [0, 1]$$

required by Ball. (Recall that we are treating the ball and bats simply as points - for an extension, see increment 3 in section 2.4.) So we have

$$
\begin{aligned}
\mathsf{Ball}(p,d) \;\triangleq\; & \mathsf{Ball}(p+d,\,d) \\
\lhd\; & p+d \in \mathsf{Court}^0 \;\rhd \\
& \mathsf{Ball}(p+(d_x,-d_y),\,(d_x,d_{-y})) \\
\lhd\; & p+d \text{ on/over } \mathsf{Wall} \;\rhd \\
& \mathsf{LX} \\
\lhd\; & p+d \text{ on/past } \mathsf{LG} \;\rhd \\
& \mathsf{RX}
\end{aligned}
\tag{9}
$$

where, with x' defined as above,

$$
\begin{aligned}
\mathsf{LX} \triangleq\; & \mathsf{lposition}?x \rightarrow (\ \mathsf{Ball}((p_x-d_x,\,p_y+d_y),\,(-d_x,\,d_y)) \\
& \lhd\ x' \text{ on or past } p_y \ \rhd \\
& \quad \mathsf{lwin} \rightarrow \mathsf{Ball}_1\ ),
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{RX} \triangleq\; & \mathsf{rposition}?x \rightarrow (\ \mathsf{Ball}((p_x-d_x,\,p_y+d_y),\,(-d_x,\,d_y)) \\
& \lhd\ x' \text{ on or past } p_y \ \rhd \\
& \quad \mathsf{rwin} \rightarrow \mathsf{Ball}_1\ ).
\end{aligned}
$$

Finally we must record the alphabet of Ball

$$\alpha(\mathsf{Ball}) \triangleq \alpha(\mathsf{Ball}_1) \cup \{\mathsf{lposition}?x : |x| \leqslant n\} \cup \{\mathsf{rposition}?x : |x| \leqslant n\}.$$

This completes the description of Ball and of PingPong.

What happens at the end of the game? If the ball penetrates say the left-hand goal, then it engages in event lwin with both players (who therafter terminate successfully) and, after returning briefly to its initial position, resumes its movement around the court. The bats remain oblivious of the left-hand player's victory, and await further communications with their players or the ball.

Exercise. Modify the specification of PingPong so that when one of the players wins, the whole game terminates successfully, to be resumed only after the event money. //

2.4. Extensions

The specification has been presented incrementally. For example a simplified version of LB (which did not communicate its state) was presented, then replaced with a more accurate one (which did). This is more a matter of style than of notation, but one which is supported by CSP.

We suggest that the incremental style is one which is genuinely useful in the task of specification and is not simply to be adopted for pedagogical purposes. Its justification in terms of abstraction and refinement appears in the next section. The delightful euphemism **maintenance** is used (though not intentionally) for this technique when the more abstract specification has already been implemented!

In this section we propose some further increments for the reader to try, of which the last two are more open and hence more ambitious.

Increment 1. The digital buttons for the left-hand player are replaced by a single analogue one which inputs a real number in the interval $[-1, 1]$ (and similarly for the right-hand player). Each bat remains capable of only the discrete jumps described in section 2.2, but an input from its player of +1 or -1 moves the bat k jumps ($1 \leq k << n$) in the appropriate direction. Modify the specifications of LP and LB appropriately.

Increment 2. The bats impart **spin** $(-\theta, 0, +\theta)$ to the ball depending on their direction at the time of impact. The ball's spin lies in

$$\{j\theta : j \in \mathbb{Z}\}.$$

Think of a reasonable "rebound formula" (incidence need no longer equal reflection) and modify the specification of Ball appropriately. You may assume that the speed of the ball is constant.

Increment 3. We are now concerned with how the game appears on a monochromatic screen. The bats and ball possess a **shape** and their states are interpreted **spatially**. Modify the specification of PingPong accordingly.

Increment 4. So far we have ignored the time between events. Make the processes in your specification *timed* by replacing all events e with pairs $e.t$ in which $t \in \mathbb{R}^{>0}$ represents the time of occurrence of e. What conditions, on traces of timed processes, might be imposed by the user to make the specification more realistic?

3. Specifications, Implementations and Development

Having specified PingPong we should now confront the problem which always faces specifiers of software: how do we know that our description captures all factors embraced in the requirements? In the absence of any way of proving equivalence between an informal description and a piece of formal text, we are compelled to:

(a) Discuss the increments as they are made with the **user** (the perpetrator of the requirements document) to ensure agreement. Not only does this enable the specifier to fill lacunae in the requirements document (for example the extremities of the goals in PingPong), but helps to keep him on the right track.

(b) State (and prove!) properties of the specification to check that they are expected by the user. For instance, PingPong is free from deadlock and will diverge if the ball never penetrates a goal. Such proofs are typically performed using laws to manipulate the CSP expressions, and by consideration of their semantics.

The specification of PingPong has been approached as an end in itself: no attempt has been made to implement it or to do any step-wise refinement. It is, like any high-level description which abstracts from those factors which determine the outcomes being described, nondeterministic. Since an implementation is evidently deterministic, refinement can be viewed as the step-wise removal of nondeterminism.

But when does one process **refine** another? There are two possible ways to answer.

(a) Give an algebraic definition of the refinement relation $\leqslant$ between processes. This can be done with no apparent reference to the semantics of a process. For example $\leqslant$ must be a partial order; moreover both P and Q must refine the less specific process $P \sqcap Q$ which behaves like either P or Q

$$P \sqcap Q \leqslant P, \quad P \sqcap Q \leqslant Q.$$

This syntactic approach offers advantages when automated assistants are used to prove refinement. However we elect not to follow it in this introduction and to take instead the following approach.

(b) Define a process in terms of its behaviour and define $P \leqslant Q$ to hold iff Q's behaviour is a special case of that of P. So far we have been describing processes by giving their alphabet and their definition in CSP, and occasionally we have noted the

sequences of events (traces) which they are prepared to perform. The set of traces of a process is prefix-closed:

$$(t \in \mathit{traces}(P) \wedge u \text{ a prefix of } t) \Rightarrow u \in \mathit{traces}(P).$$

Let us now identify P with its alphabet and (prefix-closed) set of traces

$$(\alpha(P), \mathit{traces}(P)).$$

For example the process LP defined by equation (1) has

$$\alpha(P) \triangleq \{\mathit{left}, \mathit{right}, \mathit{stay}\}$$
$$\mathit{traces}(P) \triangleq \{\mathit{left}, \mathit{right}, \mathit{stay}\}^*.$$

One particular special case of this behaviour is the novice who only pushes the move-left or stay buttons

$$\alpha(NP) \triangleq \{\mathit{left}, \mathit{right}, \mathit{stay}\}$$
$$NP = (\mathit{left} \rightarrow NP) \sqcap (\mathit{stay} \rightarrow NP)$$

to which we now ascribe the semantics

$$\alpha(NP) \triangleq \{\mathit{left}, \mathit{right}, \mathit{stay}\}$$
$$\mathit{traces}(NP) \triangleq \{\mathit{left}, \mathit{stay}\}^*.$$

The alphabet contains all three events because it records the level of abstraction at which the process is described. NP knows of the event right but never performs it: this is a different process from the one which represents a player who is not even aware that right is available to him.

Another special case of LP's behaviour is the manic player who again only pushes the move-left or stay buttons, but does so alternately starting with the move-left

$$\alpha(MP) \triangleq \{\mathit{left}, \mathit{right}, \mathit{stay}\}$$
$$MP = \mathit{left} \rightarrow \mathit{stay} \rightarrow MP$$
$$\mathit{traces}(MP) = \{\langle\rangle, \langle \mathit{left} \rangle, \langle \mathit{left}, \mathit{stay} \rangle, \langle \mathit{left}, \mathit{stay}, \mathit{left} \rangle, \ldots\}$$
$$= \{t \in \{\mathit{left}, \mathit{stay}\}^* \mid n \in \mathrm{dom}\, t \Rightarrow (n \text{ is odd} \Leftrightarrow t_n = \mathit{left})\}.$$

And of course MP's behaviour is a special case of NP's. The reason is simple:

$$traces(MP) \subseteq traces(NP) \subseteq traces(LP)$$

and leads us to identify "special case of behaviour" of processes with the inclusion of their trace sets. Thus we say that process Q **refines** process P in the **traces model** of CSP , written $P \leqslant Q$, iff

$$\alpha(P) = \alpha(Q) \quad \wedge \quad traces(Q) \subseteq traces(P).$$

This only enables us to express **safety** properties of a process: those which *may* occur - we have no way of saying that they must. For example the player who presses the left-move button and then has a heart attack

$$\alpha(HP) \triangleq \{left, right, stay\}$$
$$HP = left \rightarrow STOP$$
$$traces(HP) = \{\langle\rangle, \langle left\rangle\}$$

refines MP (and NP and LP of course). With the traces model we cannot ensure liveness of a process which refines our specification! The extreme case a the process STOP with only the empty trace which refines any process having the same alphabet

$$STOP_\alpha \triangleq (\alpha, \{\langle\rangle\}).$$

Thus in the traces model of CSP we have no way to prohibit STOP from refining MP: if the specifier requires MP, the implementor may correctly produce STOP!

It is thus more usually the case that we wish to interpret the process MP as describing behaviour which *must* eventually occur. For this a strengthening of the process model is necessary. Examples are the **readiness** and **failures** models of CSP. In these models the definition of refinement enables MP to be expressed in such a way that no process which terminates will refine it. However this leaves open the question of *how long* it is before its events occur: for that we must use a **timed** model of CSP. Then only processes which react sufficiently quickly, and then with the correct event, refine a process. But with this definition of a process we have no way to distinguish between STOP and a process having only the empty trace but which perpetuallly undergoes unobservable internal events. For this we must use a **divergences** model of CSP. Details of the failures and divergences models are given in the course; for the hierarchy of other models we refer, initially, to *[4]*.

Let us return to the traces model and refinement with respect to it. In a step-wise refinement $P_1, \ldots, P_n$ from specification P_1 to implementation P_n, we must have

$$P_1 \leqslant P_2 \leqslant \ldots \leqslant P_i \leqslant P_{i+1} \leqslant \ldots \leqslant P_n.$$

It is with the development of the processes P_i that nondeterminism is gradually removed from the specification P_1. In PingPong all nondeterminism arises from the players' choice of button: from that all else is determined. Let us see how this nondeterminism might be removed.

In reality the players' choices of button depend upon their reactions to the ball's position, and simulating this would be unnecessarily complex for our present purposes. Instead we propose the following mock development. First we take a simple version of the left-hand player

$$LP_1 \mathrel{\widehat{=}} (left \rightarrow LP_1) \sqcap (right \rightarrow LP_1) \sqcap (stay \rightarrow LP_1)$$

in which, with the choice of alphabet

$$\alpha(LP_1) \mathrel{\widehat{=}} \{left, right, stay\},$$

we have abstracted from whatever resolves the choice between the events in $\alpha(LP_1)$. The time has come to reveal it.

Suppose that the rather slow-witted left-hand player has an ally, LA, who though quicker is less coordinated; nonetheless he winks his left eye, right eye or blinks when LP is to engage in the events left, right and stay respectively. In between eye contortions he pauses to sip his favourite beverage

$$\alpha(LA) \mathrel{\widehat{=}} \{sip, lefteye, righteye, blink\}$$
$$LA \mathrel{\widehat{=}} sip \rightarrow ((lefteye \rightarrow LA) \sqcap (righteye \rightarrow LA) \sqcap (blink \rightarrow LA)).$$

Taking into account the eye movements of his ally, the left-hand player becomes

$$\alpha(LP_2) \mathrel{\widehat{=}} \alpha(LP_1) \cup \{lefteye, righteye, blink\}$$

$$LP_2 \mathrel{\widehat{=}} (lefteye \rightarrow left \rightarrow LP_2)$$
$$[\!](righteye \rightarrow right \rightarrow LP_2)$$
$$[\!](blink \rightarrow stay \rightarrow LP_2).$$

Recall that ▯ is used because the choice between constituent processes is to be made by the environment of LP_2 (in fact the ally). Our intention is that LA and LP_2 interacting in parallel should account for the nondeterminism apparent in LP_1.

Exercise. Express $LP_2 \| LA$ without ‖.

Answer. $LP_2 \| LA$ = sip → X where

```
X = ( lefteye → ( left → sip → X
                | sip → left → X )
    | righteye → ( right → sip → X
                 | sip → right → X )
    | blink → ( stay → sip → X
              | sip → stay → X ) ).  //
```

What is the relationship between LP_1 and LP_2? We cannot say that LP_2 refines LP_1 because they have different alphabets. However CSP has an operator, **abstraction** (also called hiding or concealment), which conceals certain events of a process. In this case the abstraction of LP_2 with respect to events

$$E \triangleq \{lefteye, righteye, blink\}$$

is a process, written

$$LP_2 \backslash E$$

with alphabet $\alpha(LP_2)-E$, and it behaves like LP_2 but ignoring the events in E. We thus expect

$$LP_2 \backslash E = LP_1.$$

This is true, and is a consequence of the law:

$$(P ▯ Q) \backslash E = (P \backslash E) ▯ (Q \backslash E)$$

which holds provided the events in E are the only events that can occur initially for both P and Q.

We have removed the nondeterminism from LP_1 by enlarging its alphabet to incorporate those events responsible for resolving the nondeterminism; thus

$$LP_2 \backslash E = LP_1, \quad \text{where } E = \alpha(LP_2) - \alpha(LP_1).$$

Exercise. Prove this equality. //

From this relationship we can derive the proof obligation accompanying the technique of incremental specification. Given a specification P and an increment Q of it with new events F, it has to be shown that $Q \backslash F$ refines P (in general equality is too stringent a requirement between $Q \backslash F$ and P, even though it does hold in the example above).

Exercise. Prove that $(LP_2 \parallel LA) \setminus \alpha(LA) = LP_1$.

4. Acknowledgements

I am grateful to Tony Hoare for his usual influence - for his encouragement, suggestions and corrections. Thanks also to Carroll Morgan, Karen Paliwoda and Jim Woodcock.

5. References

[1] C.A.R.Hoare,
A couple of novelties in the propositional calculus,
Journal of Symbolic Logic.

[2] C.A.R.Hoare,
Communicating Sequential Processes,
Prentice-Hall International, 1985.

[3] INMOS Limited,
occam Programming Manual,
Prentice-Hall International, 1984.

[4] E-R.Olderog and C.A.R.Hoare,
Specification-oriented semantics for communicating processes,
Technical Monograph PRG-37, Oxford University Computing Laboratory,
Programming Research Group, 1984.