A CALCULUS of TOTAL CORRECTNESS for COMMUNICATING PROCESSES.

by

C.A.R. HOARE

April 1981
May 1981 (with minor revisions).

/// means "new paragraph please"

A Calculus of Total Correctness for Communicating Processes.

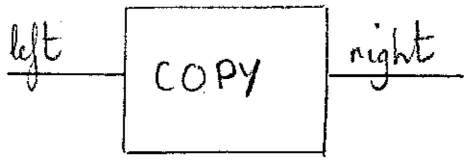
C.A.R. Hoare

April 1981

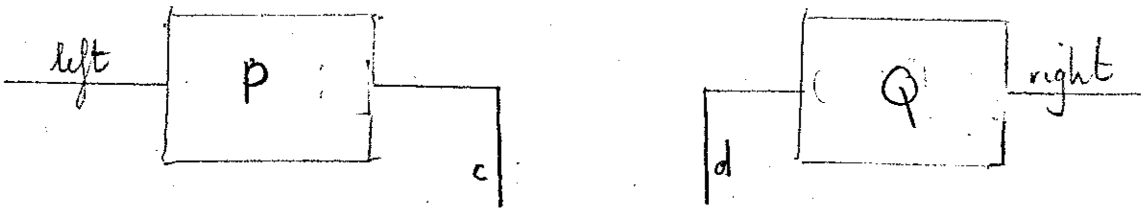
Dedication: to my son Matthew 1967–1981.

Summary.

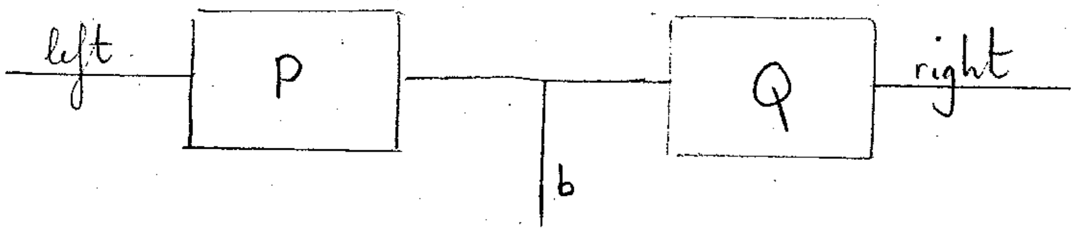
A process communicates with its environment and with other processes by synchronised output and input on named channels. The current state of a process is defined by the sequences of messages which have passed along each of the channels up to some moment in time, and also by the sets of messages that may next be passed on each channel. A process satisfies an assertion if the assertion is at all times true of all possible states of the process.
We present a calculus for proving that a process satisfies the assertion describing its intended behaviour. The following constructs are axiomatised: output; input; simple recursion; disjoint parallelism; channel renaming, connection and hiding; process chaining; nondeterminism; conditional; alternation; and mutual recursion. The calculus is illustrated by proof of a number of simple buffering protocols.



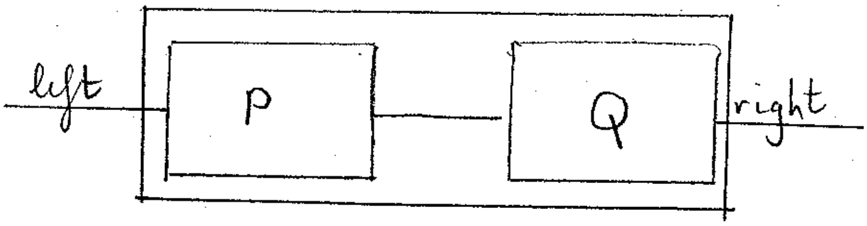
A a process with alphabet {left, right}



B a process (P ||| Q) with alphabet {left, c, d, right}



C the process (b = (c ↔ d) <u>in</u> B) with alphabet {left, b, right}



D is the process (<u>chan</u> b <u>in</u> C) with alphabet {left, right}

Figure 1

1. ASSERTIONS.

A process communicates with its environment by sending and receiving messages on named channels (Fig. 1 A). The names of these channels constitute the alphabet of the process. A process may be constructed from a group of subprocesses, intercommunicating on a network of named channels (Fig. 1 B, C). A message output by one process along a channel is received instantaneously by all other processes connected to that channel, provided that all these processes are simultaneously prepared to input that message.

close up

On each named channel, it is possible to keep a record of all messages passing along it. (For simplicity, we ignore direction of communication: if desired, this could be recorded as part of each message.) At any given moment, the record of all messages that have passed so far on a channel c is a finite sequence, which will be denoted by the variable "c.past".

no new para

At the very beginning, the value of c.past (for each channel c) is the empty sequence <>. During the evolution of a process, whenever a message m is communicated on channel c, the value of c.past is extended on the right by m, and the new value is (c.past <m>).

/// At any given moment, the set of messages which a process is prepared to communicate on channel c is denoted by the variable "c.ready." When the process is not prepared to communicate at all on channel c, the value of c.ready is the empty set $\emptyset$. When a process is prepared to input on channel c, the value of c.ready is the set M of all possible messages (for that channel). When a process is prepared to output some message value m (selected from M), then the value of c.ready is the unit set {m}, which has m as its only member.

//// Variables of the form c.past, c.ready are known as channel variables. Since we do not wish to be concerned with the internal states and transitions of a process, we shall identify the current externally observable state of a process with the current values of its channel variables.

/// An <u>assertion</u> with a given alphabet is a normal sentence of logic and mathematics, which may contain free channel variables of the form "c.past" and "c.ready", where c is a channel name in the alphabet of the assertion. The assertion describes certain possible states of some process at certain moments of time.

For example, the following are assertions, with informal explanations of their meaning.

(a) left.past = right.past

"The sequence of messages which has passed so far along the left channel is the same as the sequence that has passed along the right channel"

(b) left.ready = M

"The left channel is ready for input of any message in the set M"

(c) right.past < left.past

"The messages passed on the right channel form a proper initial subsequence of the messages that have passed on the left"

(d) right.ready = {first (left.past − right.past)}

"The right channel is ready for output of the earliest message on the left which has not yet been transmitted on the right"

/// Assertions may be readily combined by the familiar connectives of logic. For example, we define for future use the assertion:

$$\begin{aligned} \mathrm{BUFF} \triangleq\ & \mathit{left.past} = \mathit{right.past}\ \&\ \mathit{left.ready} = M \\ \vee\ & \mathit{right.past} < \mathit{left.past}\ \& \\ & \qquad \mathit{right.ready} = \{\mathrm{first}(\mathit{left.past} - \mathit{right.past})\}. \end{aligned}$$

This assertion describes all possible states of a buffering process (or transparent communications protocol), which outputs on its right channel the same sequence of messages which it inputs from the left, though possibly after some delay. When $\mathit{left.past} = \mathit{right.past}$, the process has an empty buffer, and it must then be prepared to input any message from the left. In the alternative case, the buffer is nonempty; it contains the sequence $(\mathit{left.past} - \mathit{right.past})$ of messages which are awaiting output on the right; and now the buffering process must be prepared to output the first element of this buffer. The assertion BUFF does not say whether or not input on the left is possible when the buffer is nonempty; and thus it does not specify any particular bound on the size of the buffer.

/// Let P be a process and let R be an assertion with the same alphabet as P.

Then P is said to <u>satisfy</u> R if at <u>all</u> times during the evolution of P, (before and after each communication) the assertion R correctly describes the observable state of P, i.e., the sequences of messages that have passed along its named channels, and the sets of messages that are ready to be communicated on the very next step.

This relation between processes and assertions is abbreviated:

P <u>sat</u> R.

For example any process P which is to serve as a buffer for transparent communications protocol must satisfy the assertion BUFF. There are many processes that do so; — for example, a bounded buffer of any finite size or even an unbounded buffer; examples will be given later.

It follows from the intended interpretation of the relation "satisfies" that the following properties should be true for all processes P, and all predicates, R, S.

(H1) P $\underline{\text{sat}}$ TRUE.

TRUE is a predicate which is always true of everything; it must therefore always be true of the behaviour of every process.

(H2) $\neg$ (P $\underline{\text{sat}}$ FALSE)

FALSE is the predicate that is always false of anything; it cannot therefore correctly describe the behaviour of any process.

(H3) $$\frac{R \Rightarrow S}{(P \ \underline{\text{sat}}\ R) \Rightarrow (P \ \underline{\text{sat}}\ S)}$$

If $(R \Rightarrow S)$ is a theorem, every state in which R is true is also a state in which S is true. If all states of P are correctly described by R, they must also be correctly described by S, and hence $((P \ \underline{\text{sat}}\ R) \Rightarrow (P \ \underline{\text{sat}}\ S))$ is also true. (H3) is a useful proof rule, known as the "rule of consequence"

Corollary: $$\frac{R \equiv S}{(P \ \underline{\text{sat}}\ R) \equiv (P \ \underline{\text{sat}}\ S)}$$

(H4) If n is <u>not</u> a channel variable, and does <u>not</u> occur in P:

$$(\forall n{:}N.\ P\ \underline{sat}\ R(n)) \equiv (P\ \underline{sat}\ (\forall n{:}N.\ R(n)))$$

If, for each n in some set N, P satisfies $R(n)$, then each state of P is correctly described by $R(n)$, for all n in N. The converse implication follows from (H3).

Corollary: $(P\ \underline{sat}\ R) \,\&\, (P\ \underline{sat}\ S) \equiv (P\ \underline{sat}\ (R \,\&\, S))$

These four conditions are rather similar to the healthiness conditions introduced by E.W. Dijkstra [1] to check the validity of each clause in the definition of his weakest precondition for sequential programming. Unfortunately, our calculus is not strong enough to <u>prove</u> healthiness in all cases; so we have to introduce the conditions as independent axioms, which must at least be <u>consistent</u> with the other proof rules of the calculus.

/// Let R be an assertion not containing the variable n; then we define $R \restriction n$ (R restricted to n) as the assertion satisfied by a process which behaves as described by R for at least $n-1$ steps, i.e., at least until the total number of communications on all channels reaches n. Let $\{a, \ldots, z\}$ be the alphabet of R. Let $\#s$ stand for the length of the sequence s. Then we can define:

$$R \restriction n \triangleq (\#a.past + \ldots + \#z.past \geq n) \vee R$$

Example: $BUFF \restriction n \triangleq (\#left.past + \#right.past \geq n) \vee BUFF$

Theorem 1. For any assertion R

(a) $R \restriction 0$ is a theorem

(b) $(\forall n : NAT.\ R \restriction n) \equiv R$

Proof: $c.past$ is a finite sequence for each channel c. So $\#c.past$ is a natural number. R does not contain n, so

$$(\forall n : NAT.\ R \restriction n) \equiv (\forall n. NAT.\ \#a.past + \ldots + \#z.past \geq n) \vee R$$
$$\equiv R$$

/// Let R be an assertion possibly containing a variable x, and let e be an expression of the same type as x. Then we define $R[e/x]$ as the assertion formed from R by substituting e for every free occurrence of x. (If any free variable of e would thereby become bound to a bound variable in R, the collision must be averted by systematic change of the offending bound variable.) For example, we define

$$BUFF' \triangleq (BUFF \restriction (n+1))[\langle x\rangle\, left.past / left.past]$$

$$BUFF'' \triangleq BUFF'[\langle x\rangle\, right.past / right.past]$$

After performing the substitutions, BUFF″ expands to:

$$\begin{aligned}
&\#\langle x\rangle\, left.past + \#\langle x\rangle\, right.past \geq n+1\\
\vee\ &\langle x\rangle\, left.past = \langle x\rangle\, right.past \ \&\ left.ready = M\\
\vee\ &\langle x\rangle\, right.past < \langle x\rangle\, left.past\\
&\qquad \&\ right.ready = \{first(\langle x\rangle\, left.past - \langle x\rangle\, right.past)\}
\end{aligned}$$

The following theorem is typical of the lengthy but shallow truths required in proofs of correctness of programs:

Theorem 2. $BUFF \restriction n \Rightarrow (\forall x : M.\ BUFF'')$

Proof. each clause of the LHS implies the corresponding clause on the RHS.

Let R be an assertion with alphabet a..z. We introduce the convention that

$$R[\langle\rangle/past]$$

is the result of substituting the empty sequence $\langle\rangle$ for every occurrence of any of the channel variables a.past, ..., z.past. For example

$$\begin{aligned} BUFF[\langle\rangle/past] \equiv\ & \langle\rangle = \langle\rangle \ \&\ left.ready = M \\ & \vee \langle\rangle < \langle\rangle \ \&\ \ldots \end{aligned}$$

which is equivalent to "left.ready = M". If P sat R, then $R[\langle\rangle/past]$ describes all the possible states of P at its very beginning, before it has engaged in communication on any of its channels. These states are defined in terms of a.ready, ..., z.ready, which specify the sets of communications for which P should be ready on its very first step. Thus if any process is to satisfy the assertion BUFF, it must at the beginning be ready to input on its left channel any value in the set M.

By a similar convention

$$R[\phi / ready]$$

is the result of substituting the empty set ϕ for <u>every</u> occurrence of <u>any</u> of the channel variables a.ready, ..., z.ready. For example:

$$\begin{aligned} BUFF[\phi / ready] = \; & left.past = right.past \;\&\; \phi = M \\ & \vee\; right.past < left.past \;\&\; \phi = \{\ldots\} \end{aligned}$$

which is always false. If P <u>sat</u> R, then $R[\phi / past]$ describes all possible states of P, in which it is not ready for communication along <u>any</u> of its channels. These states are known as <u>deadlock</u> states; and it is usually desired to prove that they cannot occur. The states are defined in terms of the variables a.past, ..., z.past; and therefore we only need to prove that $R[\phi / ready]$ is false for <u>all</u> values values of these variables. For example, any process that satisfies BUFF can never deadlock, (unless the set M of all possible messages is empty, — a possibility which we can realistically ignore).

As a final convention, we allow successive substitutions to be separated by commas; for example:

$$R[\langle\rangle / past, \phi / ready] = (R[\langle\rangle / past])[\phi / ready].$$

/// One of the simplest processes (with alphabet A) is the process $STOP_A$ which is already deadlocked at its start. Clearly, it is never ready to do anything, so $c.ready = \phi$ for all c in A. Furthermore, the sequence of messages transmitted along each channel remains forever empty, i.e. $c.past = \langle\rangle$. In summary, the process $STOP_A$ has only this single state; consequently, it satisfies an assertion R if and only if R correctly describes its only state, i.e if R is true when all the variables of the form $c.ready$ take the value ϕ, and all the variables of the form $c.past$ take the value $\langle\rangle$. This informal reasoning justifies the axiom:

$$(STOP_A \ \underline{sat}\ R) \equiv R[\phi/ready, \langle\rangle/past]$$

Examples. The following are theorems:

$$STOP_A \ \underline{sat}\ (c.ready \neq \{x\} \ \& \ \# c.past \neq 3)$$

$$\neg(STOP_{LR} \ \underline{sat}\ BUFF)$$

where $LR = \{left, right\}$

/// $STOP_A$ is rather a useless process; it has been introduced here only to provide a simple example of an axiom, and how it can be informally justified.

In the remainder of this monograph, we introduce a number of programming constructs suitable for the programming of communicating processes.
Each construct is given a syntax, and an informal explanation of its semantics. The semantics is formalised by an axiom or proof rule which is illustrated by application to some simple example. Treatment of each example is spread over several consecutive subsections.

2.1 Output

Let P be a process; let c be a channel name in the alphabet of P; and let e be an expression (<u>not</u> containing channel variables). Then we use the notation

$$(c!e \rightarrow P)$$

to denote the process which first outputs the value of e on channel c and then behaves like P.

In its initial state, when the past of all its channels is empty, this process is prepared to communicate the value of e on channel c, so that $c.ready = \{e\}$. It is not prepared to communicate on any other channel, so initially $d.ready = \phi$ for all channels d other than c. An assertion R is true of this initial state if and only if it is true when the channel variables of R take their initial values, as described above. This may be expressed by substituting these values in R, giving

$$R[\langle\rangle/past,\ \{e\}/c.ready,\ \phi/\langle ready\rangle]$$

(The use of the expression e to stand for its value is justified only in a programming notation which excludes assignment of new values to variables occurring in e.

The subsequent states of $(c!e \rightarrow P)$ are very similar to the states of P; the only difference is in the value of c.past. If in a state of P c.past has value s, then in the corresponding state of $(c!e \rightarrow P)$, c.past has the value $\langle e \rangle s$. In order to prove

$$(c!e \rightarrow P) \ \underline{\text{sat}}\ R$$

it is the process P that must ensure, not that its <u>own</u> states satisfy R, but rather that the corresponding states of $(c!e \rightarrow P)$ are correctly described by R. In other words, R must be true when the value of c.past is replaced by $(\langle e \rangle\, c.past)$; or more formally:

$$P \ \underline{\text{sat}}\ (R\,[\langle e \rangle c.past / c.past])$$

To prove that all states of a process are correctly described by R, it is sufficient to prove that the initial state satisfies R, and that the subsequent states do so too. The preceding paragraphs deal with these two cases; putting them together we get the rule:

$$((c!e \rightarrow P) \ \underline{\text{sat}}\ R) \equiv (\ R[\langle\rangle/past, \{e\}/c.ready, \emptyset/ready] \\ \& \ P \ \underline{\text{sat}}\ (R\,[\langle e \rangle c.past / c.past]))$$

Example.

$$((\text{right}!x \to p) \underline{\text{sat}}\ \text{BUFF}') \equiv$$
$$S \ \&(p\ \underline{\text{sat}}\ \text{BUFF}'[\langle x\rangle\,\text{right.past}/\text{right.past}])$$

where $S \triangleq \text{BUFF}'[\langle\rangle/\text{past},\ \{x\}/\text{right.ready},\ \phi/\text{ready}]$

On performing the substitutions, S expands to

$$\#\langle x\rangle + \#\langle\rangle \geq n+1$$
$$\vee\ \langle x\rangle = \langle\rangle \ \&\ \phi = M$$
$$\vee\ \langle\rangle < \langle x\rangle \ \&\ \{x\} = \{\text{first}(\langle x\rangle - \langle\rangle)\}.$$

The last clause makes S a trivial theorem.

Theorem 3.

$$((\text{right}!x \to p)\ \underline{\text{sat}}\ \text{BUFF}') \equiv (p\ \underline{\text{sat}}\ \text{BUFF}'')$$

Proof. The theorem S can be omitted from a conjunction, and the definition of BUFF'' is used.

The axiom for output has the same apparent "backwards" quality as the axiom of assignment in sequential programming. Readers who have become familiar with the latter may note that the command $(c!e \to P)$ has the same apparent effect on c.past as the command

$$(P;\ c.\text{past} := \langle e\rangle\, c.\text{past})$$

provided that P contains no assignments. Thus the second term of the axiom of output is derivable from the axiom of assignment.

2.2 Input

Let $P(x)$ be a process whose behaviour (but not alphabet) possibly depends on the value of the free variable x. Let c be a channel in the alphabet of $P(x)$, and let M be a finite ~~nonempty~~ set of message values which can be communicated on channel c. Then

$$(c?x : M \to P(x))$$

is the process which is initially prepared to input on channel c any value in the set M. The newly input value is given the local name x, and the process subsequently behaves like $P(x)$. The variable x is regarded as a bound variable, so

$$(c?x : M \to P(x))$$

is the same process as

$$(c?y : M \to P(y)).$$

Example.

$$\text{COPYSTEP} \triangleq (left?x : M \to (right!x \to p))$$

COPYSTEP first inputs a value from the left, then outputs this same value to the right, and then behaves like p.

/// The input command is similar to the output command except in two respects. Firstly, the initial value of c.ready is not just a single value, but the whole of the set M. Secondly, the subsequent behaviour $P(x)$ may depend on the input value x, which is not known in advance; and therefore $P(x)$ must be proved to meet its specification for <u>all</u> values of x ranging over the set M. This reasoning informally justifies the axiom:

Let R be an assertion <u>not</u> containing x.

$$\big((c?x{:}M \to P(x))\ \underline{\mathrm{sat}}\ R\big) \equiv \big(R[\langle\rangle/\mathrm{past},\ M/c.\mathrm{ready},\ \emptyset/\mathrm{ready}]$$

$$\&\ \forall x{:}M.\big(P(x)\ \underline{\mathrm{sat}}\ R[\langle x\rangle c.\mathrm{past}/c.\mathrm{past}]\big)\big)$$

Example.

$$(\mathrm{COPYSTEP}\ \underline{\mathrm{sat}}(\mathrm{BUFF}\upharpoonright n{+}1)) \equiv$$

$$S\ \&\big(\forall x{:}M.\ (\mathrm{right}!x \to p)\ \underline{\mathrm{sat}}\ \mathrm{BUFF}'\big)$$

where $S \triangleq (\mathrm{BUFF}\upharpoonright n{+}1)[\langle\rangle/\mathrm{past},\ M/\mathrm{left.ready},\ \emptyset/\mathrm{ready}]$

$$\equiv (\#\langle\rangle + \#\langle\rangle \geq n{+}1) \vee (\langle\rangle = \langle\rangle\ \&\ M = M) \vee (\langle\rangle < \langle\rangle\ \& \ldots)$$

The second clause makes S a theorem.

Theorem 4.

$$(\mathrm{COPYSTEP}\ \underline{\mathrm{sat}}(\mathrm{BUFF}\upharpoonright n{+}1)) \equiv \big(p\ \underline{\mathrm{sat}}(\forall x{:}M.\ \mathrm{BUFF}'')\big)$$

Proof. Theorem 3, definition of BUFF′ and (H4)

2.3. Recursion

Let p be a variable standing for a process with a given alphabet. Let $F(p)$ be the description of a process (with the same alphabet) containing none or more occurrences of the variable p. Then

$$\mu p.\ F(p)$$

is the recursively defined process, which starts off behaving like $F(p)$, and on encountering an occurrence of p, behaves like $(\mu p.\ F(p))$ again.

Example.

$$\mathrm{COPY} \triangleq \mu p.\ (\mathit{left}?x : M \rightarrow (\mathit{right}!x \rightarrow p))$$

The process COPY is an infinitely repeating cycle, each iteration of which inputs a message from the left and outputs the same message to the right.

A recursively defined process is intended to be a "fixed point" of its defining function, F, i.e.,

$$\mu p.\ F(p) \text{ is the same process as } F(\mu p.\ F(p)) \ \ldots \qquad (1)$$

Let R be an assertion, and suppose for an arbitrary process p we can prove

$$(p \ \underline{\mathrm{sat}}\ (R \restriction n)) \Rightarrow (F(p)\ \underline{\mathrm{sat}}\ R \restriction (n+1)). \quad ---- \quad (2)$$

From theorem 1(a) and (H1) it follows that

$$(\mu p.\ F(p))\ \underline{\mathrm{sat}}\ (R \restriction 0)$$

By substituting $\mu p\, F(p)$ for p in (2), and using (1) we get

$$(\mu p.\, F(p) \; \underline{sat} \; R \restriction n) \Rightarrow (\mu p.\, F(p) \; \underline{sat} \; R \restriction (n+1))$$

(close up)

By the obvious induction on n we get

$$\forall n.\, (\mu p.\, F(p) \; \underline{sat} (R \restriction n))$$

By (H4) and theorem 1(b), we conclude

$$(\mu p.\, F(p)) \; \underline{sat} \; R.$$

This reasoning serves as an informal justification of the following proof rule:

$$\frac{(p \; \underline{sat} (R \restriction n)) \Rightarrow (F(p) \; \underline{sat} (R \restriction n+1))}{\mu p.\, F(p) \; \underline{sat} \; R}$$

Theorem 5. COPY $\underline{sat}$ BUFF.

By the rule given above, it is sufficient to prove

$$(p\ \underline{sat}\ (BUFF\restriction n)) \Rightarrow (COPYSTEP\ \underline{sat}\ (BUFF\restriction n+1))$$

By theorem 4, this is equivalent to

$$(p\ \underline{sat}\ BUFF\restriction n) \Rightarrow (p\ \underline{sat}\ (\forall x : M.\ BUFF''))$$

which follows from theorem 2 by (H3).

/// Now at last we see the motivation for the choice of assertions used in the previous examples. Of course, a proof would normally be presented in the reverse order, with proof requirements for the component processes being derived by formal manipulation from the proof requirement of the whole process. The reader is invited to use this top-down method to prove the obvious fact:

$$(\mu p.\ (b!0 \rightarrow p))\ \underline{sat}\ (b.ready \neq \emptyset).$$

2.4. Channel renaming.

Let P be a process, with channel c in its alphabet, and let d be a channel name *not* in its alphabet. Then $P[d/c]$ is taken to denote a process that behaves just like P, except that

- c is removed from its alphabet
- d is included in its alphabet
- whenever P would have used channel c for input or output, $P[d/c]$ uses d instead.

$P[d/c]$ can clearly be derived from the definition of the process P by replacing each occurrence of the name c by an occurrence of d.

Example.

$$\text{COPY}[d/\text{right}] = \mu p.\,(\text{left}?x{:}M \rightarrow (d!x \rightarrow p)).$$

A similar transformation may be made to any assertion satisfied by P, in accordance with the following convention:

$$R[d/c] \triangleq R[d.\text{past}/c.\text{past},\ d.\text{ready}/c.\text{ready}].$$

The appropriate axiom is quite obvious:

$$(P[d/c]\ \underline{\text{sat}}\ R[d/c]) \equiv (P\ \underline{\text{sat}}\ R)$$

2.5. Disjoint parallelism

Let P and Q be processes with disjoint alphabets. Since they have no channel name in common, they are unconnected, and therefore cannot communicate or interact with each other in any way. The notation $(P \,|||\, Q)$ denotes a process which behaves like P and Q evolving in parallel; its alphabet is clearly the union of the alphabets of P and Q. Channel renaming can be used when needed to achieve disjointness of alphabets.

Example.

$$PROT \triangleq (COPY[d/right]) \,|||\, (COPY[c/left])$$

This combination is illustrated in Fig 1 B.

/// The states of $(P \,|||\, Q)$ correspond to elements of the cartesian product space of the set of states of P and the set of states of Q. If P satisfies S, then S has the same alphabet as P; it therefore correctly describes the current values of those channels in the state of $(P|||Q)$ which are in the alphabet of P; and hence

$$(P \,|||\, Q) \ \underline{sat}\ S.$$

Similarly, if Q $\underline{sat}$ T it follows that

$$(P \,|||\, Q) \ \underline{sat}\ T$$

Hence by (H4, corollary), we justify the proof rule.

$$\frac{(P \ \underline{sat}\ S) \ \& \ (Q \ \underline{sat}\ T)}{(P \,|||\, Q) \ \underline{sat}\ (S \& T)}$$

Example.

Let : $BUFF(c, d) \triangleq BUFF[d/right] \ \& \ BUFF[c/left]$

Theorem. 6.

$$PROT \ \underline{sat}\ BUFF(c, d)$$

Proof. immediate from Theorem 5 and the proof rules for renaming and disjoint parallelism.

2.6. Channel Connection.

Let P be a process with channels c and d in its alphabet. We may wish to connect together these two channels, so that messages passed on either of them are simultaneously passed on the other. For technical reasons, we give a new name b to the newly connected channel, and eliminate the names c and d from the alphabet of P. The process resulting from this connection and renaming will be denoted

$$(b = c \leftrightarrow d \ \underline{in}\ P).$$

Example.

$$\mathrm{PROTOC} \triangleq (b = c \leftrightarrow d \ \underline{in}\ \mathrm{PROT})$$

This is illustrated in Figure 1C. The intended effect of the connection is that everything output on the right channel by the left operand of $|||$ gets simultaneously input on the left channel of the right operand of $|||$ (and vice-versa).

~~The sequence of messages so passed is recorded as b.past, and the readiness of the joined channel is indicated by b.ready. The values of these new channel variables are defined in the follow~~

/// When two channels (c and d) are connected, a message can be passed on the connecting channel b if and only if both of the connected channels are ready for that communication; so at all times:

$$b.\mathrm{ready} = (c.\mathrm{ready} \cap d.\mathrm{ready})$$

As a consequence, whenever c is ready for output and d for input, d.ready is the universal set M, and the connected channel b ready for output of the same value as c. Similar remarks apply when d is ready for output and c for input. When both c and d are ready for input, so is b. When either of c or d is unready then so is b. There remains the case that both c and d are ready for output, and the readiness of b depends on whether the values output are the same. This case is not very useful, and should probably be excluded in a practical programming notation.

/// Each message transmitted on either of the connected channels (c and d) is instantaneously passed by the connecting channel b to the other one. The sequences of messages so transmitted must therefore always be the same:

$$b.\mathrm{past} = c.\mathrm{past} = d.\mathrm{past}.$$

/// It is the duty of an implementation of the connection operator to ensure that b.ready and b.past have the right values, as described in the above paragraphs. The programmer can just assume that this has been done. Thus we derive the proof rule

$$\frac{P \ \underline{\text{sat}}\ R}{(b = c \leftrightarrow d \ \underline{\text{in}}\ P)\ \underline{\text{sat}}\ (b.\text{ready} = c.\text{ready} \wedge d.\text{ready} \ \&\ b.\text{past} = c.\text{past} = d.\text{past} \ \&\ R)}$$

Unfortunately, the assertion in the consequent of this rule contains the channel names c and d, which are not supposed to be in the alphabet of the process concerned. This problem is easily solved by the valid technique of weakening the consequent (H3); it is easy to check that the following proof rule is a logical consequence of the one justified above.

P $\underline{\text{sat}}$ R

$$(b = c \leftrightarrow d \ \underline{\text{in}}\ P)\ \underline{\text{sat}} \Big(\exists x, y.\ \ b.\text{ready} = x \cap y$$
$$\& \ R[b.\text{past}/c.\text{past},\ b.\text{past}/d.\text{past},$$
$$x/c.\text{ready},\ y/d.\text{ready}]\Big)$$

Theorem 7.

$$\text{PROTOC}\ \underline{\text{sat}}\ \exists x, y.\ (b.\text{ready} = x \cap y \ \&\ BB)$$

where $BB \triangleq \text{BUFF}(c,d)[b.\text{past}/c.\text{past},\ b.\text{past}/d.\text{past},$
$x/c.\text{ready},\ y/d.\text{ready}]$.

Proof. immediate from theorem 6.

Here is BB written out in full:

$$(\text{left}.\text{past} = b.\text{past}\ \&\ \text{left}.\text{ready} = M$$
$$\vee\ b.\text{past} < \text{left}.\text{past}\ \&\ y = \{\text{first}(\text{left}.\text{past} - b.\text{past})\})$$
$$\&\ (b.\text{past} = \text{right}.\text{past}\ \&\ x = M$$
$$\vee\ \text{right}.\text{past} < b.\text{past}\ \&\ \text{right}.\text{ready} = \{\text{first}(b.\text{past} - \text{right}.\text{past})\}$$

2.7. Hiding

Let P be a process with channel b in its alphabet. Suppose that b is a channel which connects two or more component subprocesses of P, as described in the previous section. Since b is still in the alphabet of P, it can still be used for communication with the environment of P. Indeed, no communication can take place on channel b without the knowledge and consent of the environment. However, in the design of any mechanism, we usually wish to conceal its internal workings from its environment; and this is especially important for electronic devices, which can work millions of times faster than the environment could keep pace with. We therefore wish to hide from the environment (of P) all communications passing between subprocesses of P along channel b. Each such communication is intended to occur automatically and instantaneously as soon as all the processes connected by the channel are ready for it. And, of course, channel b must be removed from the alphabet of P. The required effect is denoted:

(<u>chan</u> b <u>in</u> P)

which declares the name b as a local channel in P.

As with other local variables, we postulate,

(chan b in P) is the same as (chan c in P[c/b])

where c is not in the alphabet of P.

Example:

PROTOCOL ≙ (chan b in PROTOC)

In this example, the channel b connects the two parallel subprocesses of the process PROTOC. One of the processes acts like a trivial transmitter of a protocol, and the other as a trivial receiver. The channel b serves as the transmission line between them. The user of the mechanism is not concerned with the nature, number, or content of the messages passing along the transmission line, which are therefore concealed from him, as shown in Fig 1D.

//// A state of the process (chan b in P) is said to be stable if there is no further possibility of communication on channel b, i.e.,

$$b.ready = \phi$$

In an unstable state, when communication is possible on channel b, we want that communication to take place invisibly at high speed; and this will bring the process to a new and usually different state. Of course, if one of the other channels is ready (and the environment is prepared to communicate on at the same time as b,) that channel, the external communication can occur instead — but (this) cannot be relied upon. If the environment is not prepared to communicate on any of the other ready channels, (we insist that a ready internal communication must sooner or later occur — and preferably sooner; Thus the unstable states are evanescent, and cannot be relied upon; in specifying the externally (visible) behaviour of processes, it seems sensible to ignore them. In other words we choose to interpret

$$P \ \underline{\text{sat}} \ R$$

as a claim that R is true of all stable states of P

For each stable state of ($\underline{\text{chan}}$ b $\underline{\text{in}}$ P), there exists a state of P in which b.ready $= \phi$ and in which b.past has some value of no further interest. This informal reasoning suggests a proof rule:

$$\frac{(P \ \underline{\text{sat}} \ R)}{(\underline{\text{chan}} \ b \ \underline{\text{in}} \ P) \ \underline{\text{sat}} \left(\exists b.past. \ R[\phi / b.ready]\right)}$$

(Here we have quantified over a channel variable as if it were an ordinary variable. The meaning is the same as if an ordinary variable s had been substituted, i.e.,

$$\exists s. \ R[s/b.past, \ \phi / b.ready])$$

/// Unfortunately this proof rule leads to a contradiction. Consider the process

$$P \triangleq \mu p.\ b!0 \rightarrow p$$

P outputs an unbounded sequence of zeroes on channel b, and is always prepared to output another; we can prove:

$$P\ \underline{sat}\ (b.ready \neq \emptyset)$$

From this, using the incorrect rule given above, we deduce

$$(\underline{chan}\ b\ \underline{in}\ P)\ \underline{sat}\ \exists b.past\,((b.ready \neq \emptyset)\,[\emptyset / b.ready])$$

The assertion here reduces to $\emptyset \neq \emptyset$, which violates the condition (H2) (counterexample due to W.A. Roscoe).

The trouble here is that we have tried to hide an <u>infinite</u> sequence of internal communications, with disastrous consequences for our theory. The consequences in practice could be equally unfortunate, because the resulting process might expend all its energies on internal communication, and never pay any further attention to its environment. This phenomenon is known as "livelock" or "infinite chatter", and there are sound theoretical and practical reasons for requiring a programmer to prove it cannot occur. A simple way of doing this is to prove that the number of messages which can be passed along the hidden channel b is bounded by some total function f of the state of the other non-hidden channels:

$$\#b.past \leq f(c.past, \ldots, z.past)$$

where $c, \ldots, z$ are all the other channels in the alphabet of the process.

Summarising the discussion above, we formulate the proof rule:

$$\frac{P \ \underline{sat}(R \,\&\, (\#b.past \leq f(c.past, \ldots, z.past))) \quad \& \ b.ready \text{ is finite}}{(\underline{chan}\ b\ \underline{in}\ P)\ \underline{sat}(\exists b.past.\ R[\emptyset / b.ready])}$$

Theorem 8.

PROTOCOL $\underline{sat}(\exists b.past, x, y. (\emptyset = x \cap y \ \& \ BB))$

Proof. $BB \Rightarrow (BB \,\&\, \#b.past \leq \#left.past)$

The conclusion follows from theorem 7 and (H3)

We are at last ready to prove

Theorem 9. PROTOCOL <u>sat</u> BUFF

Proof. We prove the assertion of Theorem 8. implies BUFF. Expanding the assertion BB we get four cases:

$$left.past = b.past = right.past \;\&\; left.ready = x = M$$
$$\vee\; right.past < b.past = left.past \;\&\; right.ready = \{first(left.past - b.past)\} \;\&\ldots$$
$$\vee\; right.past = b.past < left.past \;\&\; x = M \;\&\; y = \{\ldots\}$$
$$\vee\; right.past < b.past < left.past \;\&\; right.ready = first(b.past - right.past) \;\&\; y = \ldots$$

where irrelevant phrases are replaced by ellipses.

The first two clauses obviously imply the corresponding clauses of BUFF. The third clause describes an unstable state, and contradicts the term $(\emptyset = x \cap y)$; this case is therefore eliminated. The fourth clause also implies the corresponding clause of BUFF, using transitivity of $<$ and the fact that

$$r < b < l \Rightarrow first(b - r) = first(l - r).$$

2.8. Process chaining.

The connection of processes in a series by their right and left channels is such a useful operation that it deserves a special notation:

$$(P \Leftrightarrow Q) \triangleq \underline{\text{chan}}\ b\ \underline{\text{in}}\left(b = c \leftrightarrow d\ \underline{\text{in}}\left(\left(P[d/\text{right}]\right) ||| \left(Q[c/\text{left}]\right)\right)\right)$$

where b, c, d are fresh channel names

Example. $\quad PROTOCOL = (COPY \Leftrightarrow COPY)$

Unfortunately, the proof rule for this defined construct is hardly simpler than its definition.

Let s, x, and y be fresh variables.

Let $S' = S[s/\text{right.past}][x/\text{right.ready}]$

Let $T' = T[s/\text{left.past}][y/\text{left.ready}]$

Let f be a total function of pairs of sequences.

$$\frac{P\ \underline{\text{sat}}\ S\,,\quad Q\ \underline{\text{sat}}\ T,\qquad S' \,\&\, T' \Rightarrow s \leq f(\text{left.past}, \text{right.past})}{(P \Leftrightarrow Q)\ \underline{\text{sat}}\left(\exists s, x, y.\left(x \cap y = \emptyset \,\&\, S' \,\&\, T'\right)\right)}$$

Theorem 10 If P $\underline{\text{sat}}$ BUFF and Q $\underline{\text{sat}}$ BUFF

then $(P \Leftrightarrow Q)$ $\underline{\text{sat}}$ BUFF

Proof. essentially the same as given for theorem 9.

Corollaries: $(PROTOCOL \Leftrightarrow COPY)$ $\underline{\text{sat}}$ BUFF

$(PROTOCOL \Leftrightarrow PROTOCOL)$ $\underline{\text{sat}}$ BUFF

etc.

2.9. Nondeterministic union.

Let P and Q be process descriptions with the same alphabet. Then the notation

$$(P \underline{\text{or}}\ Q)$$

stands for a process that behaves either like P or like Q. The choice between the alternatives is left completely unspecified, and may be made arbitrarily as the process $(P \underline{\text{or}}\ Q)$ evolves, or may be fixed by its implementor before the starts. The choice cannot be influenced by the environment of the process, and is undetectable at the time it is made — though it may be deducible from the subsequent behaviour of the process.

Example. $(\text{PROTOCOL}\ \underline{\text{or}}\ \text{COPY})$

This behaves either like a one-place buffer or a two-place buffer, the choice being unspecified. and unknown. If, during the life of this process, the length of left.past ever exceeds the length of right.past by two, then we can deduce that the choice has fallen on the first alternative.

If we want to be sure that $(P \underline{\text{or}}\ Q)$ satisfies R, since we do not know which of P or Q will be selected, we had better prove that they both satisfy R

$$(P \underline{\text{or}}\ Q)\ \underline{\text{sat}}\ R \ \equiv\ (P\ \underline{\text{sat}}\ R) \,\&\, (Q\ \underline{\text{sat}}\ R)$$

Theorem 11. $(\text{PROTOCOL}\ \underline{\text{or}}\ \text{COPY})\ \underline{\text{sat}}\ \text{BUFF}$

Proof. From Theorems 9 and 5.

2.10 Conditional

Let e be a Boolean-valued expression <u>not</u> containing any channel variables. Let P and Q be processes with the same alphabet. Then the process

$$\underline{\text{if}}\ e\ \underline{\text{then}}\ P\ \underline{\text{else}}\ Q$$

is one that behaves like P if e evaluates to true, or behaves like Q if e evaluates to false. The proof rule is correspondingly simple:

$$((\underline{\text{if}}\ e\ \underline{\text{then}}\ P\ \underline{\text{else}}\ Q)\ \underline{\text{sat}}\ R)$$

$$\equiv\ \underline{\text{if}}\ e\ \underline{\text{then}}\ (P\ \underline{\text{sat}}\ R)\ \underline{\text{else}}\ (Q\ \underline{\text{sat}}\ R)$$

An example will be given in 2.12.

2.11. Alternation.

Let $P(x)$ and $Q(y)$ be processes whose behaviour possibly depends on the values of the free variables x and y respectively; but all of them have the same alphabet. Let c and d be distinct channel names in this alphabet. Let M be the set of messages that can be communicated on c, and let N be the set for d. Then the notation

$$(c?x:M \to P(x) \;[]\; d?y:N \to Q(y))$$

denotes a process which behaves as follows. Initially, it is prepared to input either on channel c or on channel d; in the first case its subsequent behaviour is defined by $P(x)$, where x stands for the value input on c; and in the second case, its subsequent behaviour is ~~similarly~~ defined by $Q(y)$, where y is the value input on d.)

(Only one of the two inputs can take place; but in contrast to nondeterministic union,) (the choice can be influenced by the other processes connected to the channels c and d. If the process (or processes) connected to one of them remains forever unprepared for communication, then communication can still occur, but only on the other channel. But if (all) the processes connected to each of the channels all become ready) (for communication, then it is nondeterministic ~~on which~~

on which channel the communication will take place. (An efficient implementation should select the first to become ready; but such considerations of efficiency rightly cannot be formalised in a calculus of correctness; and a programmer clearly must not rely on them, since he has delegated to the implementor all control over the relative speeds of the processes.

Example.

$$\begin{aligned} \text{MERGESTEP} \triangleq (&\text{left1}?x : M \to \text{right}!(1,x) \to p \\ \square\, &\text{left2}?x : M \to \text{right}!(2,x) \to p) \end{aligned}$$

This process has alphabet {left1, left2, right}. It inputs a message x on either left1 or left2, tags it with a 1 or 2 to indicate its source, and outputs the tagged message on the right, after which it behaves like p.

In the initial state of a process described using $\square$, both the channels involved are ready for input, and all the other channels are unready. Each subsequent state corresponds either to a state of $P(x)$ or to a state of $Q(y)$; and <u>both</u> cases must be proved correct. The proof rule is therefore modelled on that for simple input.

If c and d are distinct channel names:

$$(c?x{:}M \rightarrow P(x) \;[]\; d?y{:}N \rightarrow Q(y)) \;\underline{\text{sat}}\; R$$

$$\equiv \quad R[\langle\rangle/\text{past},\ M/c.\text{ready},\ N/d.\text{ready},\ \emptyset/\text{ready}]$$
$$\&\ \forall x{:}M.\ P(x)\ \underline{\text{sat}}\ R[\langle x\rangle\, c.\text{past}/c.\text{past}]$$
$$\&\ \forall y{:}N.\ Q(y)\ \underline{\text{sat}}\ R[\langle y\rangle\, d.\text{past}/d.\text{past}].$$

Example.

Let $sel(n,s)$ be a sequence formed from s by selecting only those items tagged with n, and then removing the tags; or, more formally:

$$sel(n,s) = \underline{\text{if}}\ s = \langle\rangle\ \underline{\text{then}}\ \langle\rangle$$
$$\underline{\text{else}}\ \underline{\text{if}}\ first(s) = (n,x)\ \underline{\text{then}}\ \langle x\rangle\, sel(n, rest(s))$$
$$\underline{\text{else}}\ sel(n, rest(s)).$$

if le

Let MERGED $\triangleq$ $sel(1, right.past) \leq left1.past$

$\&\ sel(2, right.past) \leq left2.past$

$\&\ (left1.ready = left2.ready = M\ \&\ EQ$

$EQ = sel(1, right.past) = left1.past$
$\&\ sel(2, right.past) = left2.past$ $\vee\ right.ready \neq \emptyset)\ \&\ \neg EQ$

Theorem 12. MERGESTEP $\underline{\text{sat}}$ $(\text{MERGED} \upharpoonright n+1) \equiv$

$$\forall x{:}M.\ (right!(1,x) \rightarrow p)\ \underline{\text{sat}}\ (\text{MERGED}\upharpoonright n+1)[\langle x\rangle\, left1.past/left1.past]$$
$$\&\ \forall x{:}M\ (right!(2,x) \rightarrow p)\ \underline{\text{sat}}\ (\text{MERGED}\upharpoonright n+1)[\langle x\rangle\, left2.past/left2.past]$$

Proof. The omitted terms are trivial theorems.

The reader may care to complete the proof that

$$(\mu p.\ \text{MERGESTEP})\ \underline{\text{sat}}\ \text{MERGED}.$$

/// The notation and proof rule for alternation can clearly be adapted for more than two alternatives; and since $(c!e \rightarrow P)$ is the same as $(c?x{:}\{e\} \rightarrow P)$, output can be readily substituted for input.

2.12 General Recursion.

The method of defining processes by recursion can be generalised to allow mutual recursion. A set of processes defined by mutual recursion constitute a solution to a set of simultaneous fixed point equations, just as $\mu p. F(p)$ is a solution for p in the single equation:

$$p \triangleq F(p).$$

A pair of mutually recursive equations take the form

$$p \triangleq F(p,q)$$

$$q \triangleq G(p,q)$$

where $F(p,q)$ and $G(p,q)$ are process descriptions, which in general contain the process variables p and q.

The method of mutual recursion generalises even further to infinite sets of simultaneous equations, one for each member s in some counting set S:

$$p(s) = F(p, s) \qquad \text{for all } s \text{ in } S.$$

The solutions to all these simultaneous equations constitute an array p, with an element $p(s)$ for each s in S. This array of processes is denoted by the formula:

$$\mu p(s{:}S).\ F(p,s).$$

However, it is often clearer to write the definitions in the equational form shown above

Example.

Let M^* be the set of all finite sequences of elements of M.

Let $IN \triangleq (left?x{:}M \rightarrow p(\langle x\rangle))$

Let $INOROUT \triangleq (left?x{:}M \rightarrow p(s\langle x\rangle)$
$\quad\quad\quad\quad\quad\quad\quad\quad \square\ right!first(s) \rightarrow p(rest(s))$
$\quad\quad\quad\quad\quad\quad\quad\quad)$

Let $STEP \triangleq$ if $s = \langle\rangle$ <u>then</u> IN <u>else</u> $INOROUT$

Let $B \triangleq \mu p(s{:}M^*).\ STEP$

For each s in M^*, $B(s)$ behaves like an unbounded buffer with current content s. If s is empty, $B(s)$ is prepared only to input on the left any value x in M, and then behave like $B(\langle x\rangle)$, that is, like a buffer containing only the value x. But if s is nonempty, $B(s)$ is prepared: newline <u>either</u> (1) to input a new element x, which is appended to the stored buffer, so that its subsequent behaviour is $B(s\langle x\rangle)$; <u>or</u> (2) to output the first element of its buffer, which is then removed, so that its subsequent behaviour is $B(rest(s))$.

The same definition can be written out more clearly in the form of an equation in B:

$$
\begin{aligned}
B(s) \triangleq \text{ if } s = \langle\rangle\ &\underline{\text{then}}\ left?x{:}M \rightarrow B(\langle x\rangle)\\
&\underline{\text{else}}\ (left?x{:}M \rightarrow B(s\langle x\rangle)\\
&\quad\ \ \square\ right!first(s) \rightarrow B(rest(s))\\
&\quad\ \ )
\end{aligned}
$$

For all s in M^*

The proof rule for generalised recursion is similar to that for simple recursion, except that the formulae are quantified over all s in the counting set S.

$$\frac{(\forall s{:}S.\ p(s)\ \underline{sat}\ (R(s)\restriction n) \Rightarrow \forall s{:}S.\ F(p,s)\ \underline{sat}\ (R(s)\restriction n{+}1))}{\forall s{:}S \quad ((\mu p(s{:}S)\,F(p,s))\,(s))\ \underline{sat}\ R(s)}$$

Example. Let us define

BUFF(s) $\triangleq$ BUFF$[s\ left.past/left.past]$

BUFF(s) describes the behaviour of a buffer that has input the sequence s, but not yet output it. BUFF(s) therefore should describe the future behaviour of the process $B(s)$, as stated in the following theorem.

Theorem 13. $\forall s: S.\ B(s)\ \underline{sat}\ BUFF(s)$

Proof.

By the rule of recursion, we can assume

$$\forall s: M^*.\ p(s)\ \underline{sat}\ (BUFF(s) \upharpoonright n) \quad \text{-- -- -- --} \quad (0)$$

and must prove

$$STEP\ \underline{sat}\ (BUFF(s) \upharpoonright n+1) \qquad \text{for } s \in M^*$$

which by the conditional rule, splits in two:

$$s = \langle\rangle \Rightarrow IN\ \underline{sat}\ (BUFF(s) \upharpoonright n+1) \quad \text{-- -- --} \quad (1)$$

$$\text{and } s \neq \langle\rangle \Rightarrow INOROUT\ \underline{sat}\ (BUFF(s) \upharpoonright n+1) \quad \text{-- -- --} \quad (2)$$

For (1), we assume $s = \langle\rangle$ and need to prove

$$(BUFF(s) \upharpoonright n+1)[\langle\rangle/past,\ M/left.ready,\ \phi/ready] \quad \text{-- -- --} (1a)$$

$$p(\langle x\rangle)\ \underline{sat}(BUFF(s) \upharpoonright n+1)\ [\langle x\rangle left.past/left.past] \quad \text{-- -- --} (1b)$$

(1a) is a trivial theorem, and the assertion of (1b) is equivalent to

$p(\langle x\rangle)\ \underline{sat}\ BUFF[s\langle x\rangle left.past/left.past] \upharpoonright n$

which by definition is $BUFF(s\langle x\rangle) \upharpoonright n$

So (1b) follows directly from the assumption (0) and the condition $s = \langle\rangle$.

For (2) we assume $s \neq \langle\rangle$ and need to prove

$$(BUFF(s) \upharpoonright n+1)[\langle\rangle/past,\ M/left.ready,\ \{first(s)\}/right.ready,\ \phi/ready] \quad (2a)$$

$$\&\ \forall x: M.\ p(s\langle x\rangle)\ \underline{sat}\ (BUFF(s) \upharpoonright n+1)\ [\langle x\rangle left.past/left.past] \quad (2b)$$

$$\&\ p(rest(x))\ \underline{sat}\ (BUFF(s) \upharpoonright n+1)[\langle first(s)\rangle right.past/right.past] \quad (2c)$$

(2a) is a trivial theorem. The assertion of (2b) is equivalent to $BUFF(s\langle x\rangle) \upharpoonright n$, and the assertion of (2c) is equivalent to $BUFF(rest(s)) \upharpoonright n$; so both (2b) and (2c) follow from the assumption (0)

/// To check the above claims of trivial theoremhood or equivalence, it is necessary only to expand the abbreviations. For example

$$(\mathrm{BUFF}(s) \restriction n+1)[\langle \mathrm{first}(s) \rangle \, \mathrm{right.past} / \mathrm{right.past}] \equiv$$

$$\begin{aligned} \equiv \quad & \# \mathrm{left.past} + \# (\langle \mathrm{first}(s) \rangle \, \mathrm{right.past}) \geq n+1 \\ & \vee (s \; \mathrm{left.past}) = \langle \mathrm{first}(s) \rangle \, \mathrm{right.past} \; \& \; \mathrm{left.ready} = M \\ & \vee \langle \mathrm{first}(s) \rangle \, \mathrm{right.past} < (s \; \mathrm{left.past}) \\ & \qquad \& \; \mathrm{right.ready} = \{ \mathrm{first}(s \; \mathrm{left.past} - \langle \mathrm{first}(s) \rangle \, \mathrm{right.past}) \} \end{aligned}$$

$$\mathrm{BUFF}(\mathrm{rest}(s)) \restriction n \quad \equiv$$

$$\begin{aligned} & \# \mathrm{left.past} + \# \mathrm{right.past} \geq n \\ & \vee (\mathrm{rest}(s) \; \mathrm{left.past}) = \mathrm{right.past} \; \& \; \mathrm{left.ready} = M \\ & \vee \mathrm{right.past} < (\mathrm{rest}(s) \; \mathrm{left.past}) \\ & \qquad \& \; \mathrm{right.ready} = \{ \mathrm{first}((\mathrm{rest}(s) \; \mathrm{left.past}) - \mathrm{right.past}) \} \end{aligned}$$

When $s \neq \langle\rangle$, these are clearly equivalent, clause by clause.

Theorem 14. $B(\langle\rangle)$ <u>sat</u> BUFF

Proof. Put $s = \langle\rangle$ in Theorem 13.