SPECIFICATION, PROGRAMMING AND VERIFICATION OF CSP BY CHANNEL OBSERVATION SEQUENCES

M. G. Gouda and H. S. Shen

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-227 May 1983

Table of Contents

1. INTRODUCTION 1
2. SPECIFICATIONS AND PROGRAMS 2
3. A PROGRAMMING LANGUAGE 2
4. PROGRAM EXAMPLES 4
5. LANGUAGE PROPERTIES 6
6. A PROGRAMMING METHODOLOGY 6
7. PROVING CHANNEL LIVENESS 7
8. CONCLUDING REMARKS 11
APPENDIX:PROOFS OF LEMMAS 14

ABSTRACT

We propose to specify (i.e., state requirements) and program (i.e., define) systems of CSP using sets of finite strings called channel observation sequences. Each sequence is a concatenation of some channel symbols that defines a possible order of communications across different channels in the system under consideration. We present a simple programming language to define CSP systems in this framework; and show that the language satisfies some nice algebraic properties. We discuss a programming methodology to define CSP systems that meet given specifications. We also discuss a verification methodology to establish channel "liveness" in given CSP systems. A number of CSP systems (e.g., a finite buffer, a factorial multiplier, and a circulating token) are defined and verfied using this approach.

"We aim to describe a concurrent system fully enough to determine exactly what behaviour will be seen or experienced by an external observer... two systems are indistinguishable if we cannot tell them apart without pulling them apart." *R. Milner,1980[8]*

"In a special purpose language, such a limitation may be an advantage, if it permits a mechanical check against certain undesirable occurrences such as non-termination or deadlock."
C.A.R. Hoare,1982[5]

1. INTRODUCTION

Hoare[5] has presented a framework to specify (i.e., state requirements) and program (i.e., define) systems of CSP[3] based entirely on predicates. In this framework, if the requirements of a system are defined by a predicate S and if a CSP system is defined by a predicate P, then the veification problem of whether the defined system satisfies the specified requirements is reduced to the problem of whether predicate P logically implies predicate S. Hehner and Hoare[2] have demonstrated that a verification system based on this framework can be more powerful than a traditional verification system[4] based on defining axioms or proof rules for each CSP construct.

In this paper, we present a framework, parallel to that of Hoare, where systems of CSP can be both specified and defined using sets of channel symbol sequences. If the requirements of a system are specified by a set S, and if a CSP system is defined by a set P, then whether the defined system meets the specified requirements is reduced to whether P is a subset of S. Our approach seems to cure two minor problems in Hoare's framework. First, the sometime unexpected behaviour of the sequencing or "first action" operator is no longer present. Second, the restriction that the "alternation" operator should be applied to predicates with the same sets of variables is no longer needed.

We also extend our framework to support a verification methodology to prove that some channels in a CSP system are "live"; i.e., communications across these channels are guaranteed to occur infinitely often. The verification methodology is based on some simple decision procedures for regular expressions and formal languages.

Defining systems of CSP using sets of channel symbol sequences has its roots in the works of Milner[8], and Merlin and Bochmann[9]. The pioneering work of Milner[8] presents a general and extensive calculus to define and verify systems of communicating processes, not necessarily CSP systems. Processes in Milner's calculus can be nondeterministic, and can exchange data values during their communications. By contrast, we restrict our attention in this paper to deterministic processes which exchange only uninterpreted messages. These restrictions have led to a different and simpler programming language to define CSP systems, which in turn led to simpler programming and verification methodologies than those offered in [8].

Merlin and Bochmann[7] have proposed to define both the service and the constituent processes of a protocol layer using finite state models. They, and later Gouda and Chu[1], have demonstrated that some synthesis problems of communication protocols can be modeled and solved easily in this framework.

The paper is organized as follows. Section 2 defines our basic characterization of "specifications" and "programs" for CSP systems. Based on this characterization, a simple programming language for CSP is presented in Section 3; and some program examples are given in Section 4. Some "nice" properties of our language are discussed in Section 5. Then, a programming methodology based on our language is presented in Section 6; and a verification methodology to establish channel liveness in programs of our

language is discussed in section 7. Concluding remarks are in Section 8. For convenience, all the lemmas are proved in the Appendix.

2. SPECIFICATIONS AND PROGRAMS

Let A={a,b,...} be a finite set of symbols called *channel symbols*; A is called the *channel alphabet*.

A *specification* S (of some CSP systems) is a nonempty set, of finite strings over the channel symbols in A, which is closed under prefixing; i.e., if a string s is in S, then any prefix of s is also in S. A string in S is called a *channel observation sequence* (or simply a *sequence*) of S. The set of symbols used in the sequences of S is a subset of the channel alphabet A; it is denoted @S.

Example: Consider a system that consists of one sequential process and two channels called "left" and "right". Repeatedly, the process receives a message via channel "left" then sends a message via channel "right". This system can be specified by the infinite set of channel observation sequences:

{E,
left,
left.right,
left.right.left,...},

where E denotes the empty string, and
"." is the usual concatenation operator.

Notice that each channel observation sequence defines a possible limited history of the process as it accesses the two channels. []

A *program* P is an expression, written in some restricted notation called a *programming language*, which defines a closed-under-prefixing set of finite strings over the channel symbols in A. From now on, we use the term "a program" to mean an expression which defines a set or to mean the set itself; the context should indicate, in each instance, which usage is intended.

Each string in a program P is called a *channel observation sequence* (or simply a *sequence*) of P; and @P denotes the set of channel symbols used in the sequences of P.

A program P is said to *meet* a specification S iff P is a subset of S (denoted $P \Rightarrow S$).

3. A PROGRAMMING LANGUAGE

In this section, we present a simple language to define sets of strings over the channel alphabet A. Later, in section 5, we show that each set defined by this language is closed under prefixing; and so the language is a programming language.

In our language, a set of strings over the channel alphabet A is defined by an *expression*:

E(P, ... ,Q)

which contains some symbols P,...,Q, called *program symbols*. These program symbols are defined by some *recursive equations*:

P = F(P), ... , Q = G(Q)

where F(P) is an expression which contains the program symbol P, and G(Q) is an expression which contains the program symbol Q. Beside their program symbols, the expressions E, F,..., and G may also contain any of two constants and three operators. The two constants are denoted "ZERO" and "CHAOS"; and the three operators are denoted "a;", "v", and "&". They are defined as follows:

Constant ZERO: ZERO is the set {E}, where E is the empty string. []

Constant CHAOS: CHAOS is the set A^*, where A is the channel alphabet and * is the usual transitive closure operator. []

Operator "a;": Let "a" be a channel symbol in A; and let R be any set of setrings over the channel symbols in A. Then a;R is the set {E} ∪ a.R, where ∪ is the usual union operator, and a.R is the set of all strings a.s, where s is a string in R. []

Operator "v": Let R and R' be two sets of strings over the channel symbols in A. Then, R v R' is the set R ∪ R'. []

Operator "&": Let s and s' be two strings over the channel symbols in A; and let B be a subset of A which contains all the common symbols in s and s'. Then, s and s' are *compatible with respect to* B iff they have the same symbols of B in the same order; i.e., they can be written as:

$s = x_0.a_1.x_1. \ \dots \ .a_k.x_k$, and

$s' = y_0.a_1.y_1. \ \dots \ .a_k.y_k$,

where $a_1,\dots,a_k$ are symbols in B, and $x_0,\dots, x_k, y_0,\dots,y_k$ are strings over the symbols in A - B.

Let s and s' be two compatible strings with respect to some set of channel symbols which contains all the common symbols in s and s'. As defined above, s and s' can be written as:

$s = x_0.a_1.x_1. \ \dots \ .a_k.x_k$, and

$s' = y_0.a_1.y_1. \ \dots \ .a_k.y_k$,

where $a_1,\dots,a_k$ are the common symbols in s and s'. Define s&s' as the set:

$\text{intrlv}(x_0,y_0).a_1.\text{intrlv}(x_1,y_1). \ \dots \ .a_k.\text{intrlv}(x_k,y_k)$,

where $\text{intrlv}(x_i,y_i)$ is the set of all strings constructed by interleaving the symbols of x_i with the symbols of y_i.

Let R and R' be two sets of strings over the channel symbols in A; then R&R' is the set:

```
    ∪    (s&s')
s is in R,
s' is in R', and
s and s' are compatible with respect to @R∩@R'
```

There is another useful characterization of the "&" operator. Let s be a string over the channel

symbols in A; and let B be a subset of A. The *projection* of s *over* B, denoted $\#_B s$, is a string obtained from s by replacing each channel symbol in s but not in B by the empty string. (For example, if s = abccbabacb and B = {a,c}, then $\#_B s$ = accaac.)

Lemma 1: Let R and R' be two sets of strings over the channel symbols in A; and let s be a string over the channel symbols in A.

s is in R&R'

iff $\#_{@R} s$ is in R, $\#_{@R'} s$ is in R', and $\#_{@R \cup @R'} s = s$. []

As mentioned earlier, the above constants and operators can be used to define the expressions E(P,...,Q), F(P), and G(Q). It remains now to discuss the meaning of "recursion" defined by the equations P = F(P) and Q = G(Q).

Recursion: An equation P = F(P), where P is a symbol and F(P) is an expression , defines P as the *largest* set of strings (over the channel symbols in the channel alphabet A) which satisfies the equation. In other words, P denotes the "weakest" solution of the equation. Later (in Lemma 6 below), we show that each equation has a solution, and that there is a systematic technique to find this solution. []

4. PROGRAM EXAMPLES

Each of the following examples is written in terms of some parameter N; therefore we use subscripted program symbols (e.g., P_i, $CELL_i$) and subscripted channel symbols (e.g., t_i, $flow_i$), where the subscript i ranges over the domain 1..N in most cases. We also use the following familiar syntax:

$\mathop{\&}\limits_{i=1}^{N} P_i$ to mean $P_1 \& P_2 \& \ldots \& P_N$, and

$i=1..N$: $P_i=F(P_i)$ to mean $P_1=F(P_1), \ldots, P_N=F(P_N)$

Example (A Finite Buffer): A finite buffer is placed between two processes named SOURCE and SINK. The buffer consists of N processes named $CELL_1,\ldots,CELL_N$. Each $CELL_i$ can store up to one element and is connected to its neighbours by two channels named $flow_i$ and $flow_{i+1}$ as shown in Figure 1a. The system can be defined in our language as follows:

$SOURCE \ \& \ (\mathop{\&}\limits_{i=1}^{N} CELL_i) \ \& \ SINK$

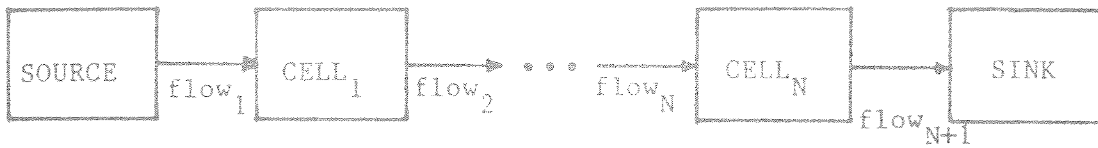
$SOURCE = flow_1;SOURCE$

$i=1...N$: $CELL_i=flow_i;(flow_{i+1};CELL_i)$

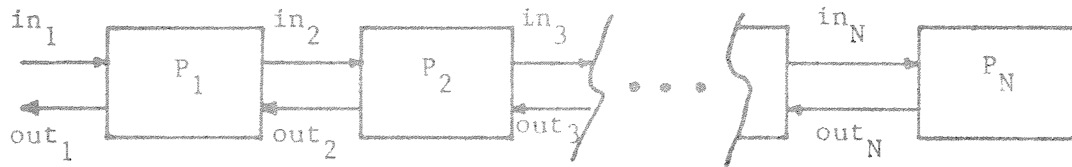
$SINK = flow_{N+1};SINK$

Notice that process names are used as program symbols and channel names are used as channel symbols. []

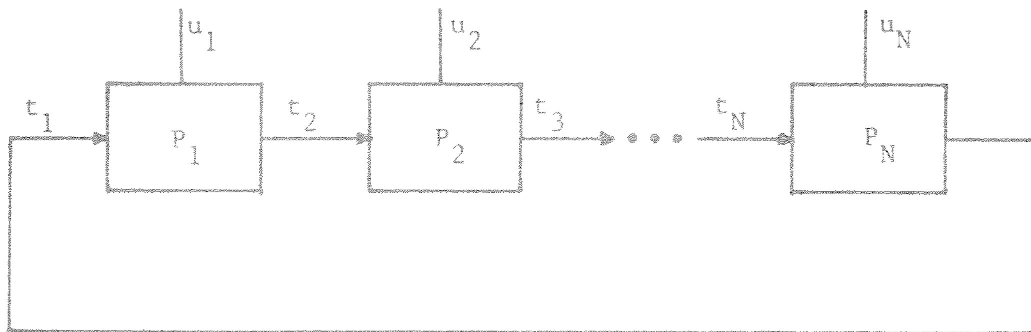
Example (A Factorial Multipler): We consider the factorial multipler proposed by Hoare[3]; it



(a) A finite buffer



(b) A factorial multiplier



(c) A circulating token

Figure 1. Examples

computes n! for any n, $1 \le n \le N$. The multipler consists of N processes named $P_1,...,P_N$. For i=1,...,N-1, P_i has two input channels in_i and out_{i+1}, and two output channels in_{i+1} and out_i. P_N has one input channel in_N and one output channel out_N. The arrangement is as shown in Figure 1b.

At the beginning, each P_i is waiting to receive a number via in_i. If the received number is 1, then P_i returns 1 via out_i and returns to its initial state. If the received number is k (k>1), then P_i keeps k and sends k-1 via in_{i+1} then waits to receive a number via out_{i+1}. When this number arrives, P_i multiplies it with the stored number k, sends the result via out_i, and returns to its initial state. This system can be defined in our language as follows:

$$\mathop{\&}_{i=1}^{N} P_i$$

$$i=1...N-1:\ P_i=(in_i;(out_i;P_i))\ V\ (in_i;(in_{i+1};(out_{i+1};(out_i;P_i))))$$

$$P_N=in_N;(out_N;P_N)$$

Notice that our language cannot model the internal workings of processes; it merely models their external (observable) behaviour. []

Example (A Circulating Token): Consider the system in Figure 1c. It is structured as a cycle of N processes P_i, and N channels t_i, i=1,...,N. Also, each process P_i has a channel u_i to communicate with the environment. At the beginning, process P_1 "has the token". When a process P_i has the token, it communicates with the environment via u_i, then sends the token to the next process P_{i+1} via t_{i+1}. (If the process is P_N, then its next process is P_1). The system can be defined in our language as follows:

$$\mathop{\&}_{i=1}^{N} P_i$$

$$P_1=u_1;(t_2;(t_1;P_1))$$

$$i=2...N-1:\ P_i=t_i;(u_i;(t_{i+1};P_i))$$

$$P_N=t_N;(u_N;(t_1;P_N))$$

[]

Other Examples: Other examples from Hoare[3] can be defined using our language; they include the integer semaphore, the dining philosophers, and the matrix muitiplication array. Unfortunately, there are other examples, also from Hoare[3], that cannot be defined using our simple language. One such example is the small set of integers. The reason that our language cannot define this example is that the allowed recursion in our language is not powerful enough. To be able to define this set of integers and the remaining examples in Hoare[3], our language should be extended to allow programs of the form

$E(P,...,Q)$

$P=F(P,...,Q)$

:

$Q=G(P,...,Q)$.

This extension is not discussed any further in this paper.

5. LANGUAGE PROPERTIES

In this section, we first prove some nice algebraic properties for the language operators. Then, we show that each expression in the language defines a set of strings that is closed under prefixing; this implies that the language is a programming language.

Algebraic Properties

Lemma 2: The operator "a;" distributes through"v". []

Lemma 3: The operator "v" is idempotent, commutative, associative, and distributes through "v". []

Lemma 4: The operator "&" is idempotent, commutative, associative, and distributes through "v". []

Closure Under Prefixing

The following four lemmas establish that each expression in our language defines a set (of strings over the channel symbols in A) which is closed under prefixing.

Lemma 5: Each expression E(P,...,Q) defines a set of strings over the symbols in A provided that P,..., and Q are sets of strings over the symbols in A. []

Lemma 6; namely: Each equation $P = F(P)$ has a solution; namely:

$P = \lim n \rightarrow \infty . F^n(A^*)$,

where $F^0(A^*)=A^*$, and

$F^n(A^*)=F(F^{n-1}(A^*))$ for $n=1,2,...$

This solution defines a set of strings over the symbols in A. []

Lemma 7: The set defined by E(P,...,Q) is closed under prefixing provided that P,..., and Q denote string sets which are closed under prefixing. []

Lemma 8: The solution $P=\lim n \rightarrow \infty . F^n(A^*)$ of equation $P=F(P)$ defines a set of strings which is closed under prefixing. []

Because of these four lemmas, our language can be called a programming language; and each expression, in our language, can be called a program. Notice that Lemma 6 suggests a systematic technique to solve any equation $P=F(P)$. First, compute the set of strings $F^n(A^*)$ in terms of n; then take the limit as n approaches infinity to get P. For example, consider the equation $P=a;P$. $F(A^*)=E+aA^*$; $F^2(A^*)=E+a+aaA^*$,..., $F^n=E+a+a^2+...+a^{n-1}+a^nA^*$. Taking the limit as n approaches infinity, we get the solution $P=a^*$.

6. A PROGRAMMING METHODOLOGY

The task of programming a specification S consists of defining a program P, in some programming language, such that P is a subset of S, i.e., $P \Rightarrow S$. To aid the programmer in this task, we propose the following two-step methodology to construct a program, in our language, that meets an arbitrary specification.

First Step (Divide and Conquer):

i. Let S be the given specification.

ii. Find an expression E(P,...,Q) based on the program symbols P,..., and Q and find some specifications (not necessarily programs) U,..., and V such that
$E(U,...,V) \Rightarrow S$
(Recall that operators in our language, and so expressions, can be applied to specifications as well as to programs.)

iii. Now the problem of finding a program to meet S has been reduced to finding some programs P,..., and Q which meet U,..., and V respectively; i.e., $P \Rightarrow U$,..., and $Q \Rightarrow V$. Each of the programs P,..., and Q can be constructed by introducing recursion as discussed in the next step.

Second Step (Introduce Recursion):

i. Let U be the given specification.

ii. Find an expression F(P) based on the program symbol P such that
$F(U \vee S_n) \Rightarrow U \vee S_{n+1}$
where S_n is the set of all finite strings, of length n or more, that are based on the channel symbols in A. (Operators in our language, and so expressions, can be applied to arbitrary sets of strings not necessarily programs or specifications).

iii. If such an F(P) can be found, then the solution of the recursive equation P=F(P) is a subset of U; i.e., $P \Rightarrow U$. []

Lemma 9: The proposed programming methodology is correct. []

Example: Let the channel alphabet be A={a,b,c,d}; it is required to define a program in our language to meet the following specification (defined as a regular expression):

$S = a^* + b^* + ab + c + cd$

Define the following expression in our language:

E(P,Q) = P v Q v (a;(b;ZERO)) v (c;(d;(ZERO)).

Also define the following two specifications U and V (defined as regular expression): $U=a^*$, and $V=b^*$. Since E(U,V) is a subset of S, it remains now to define two programs P and Q which meet U and V respectively. Define the following expression in our language:

F(P) = a;P

Since $F(U \vee S_n) \Rightarrow U \vee S_{n+1}$, then the solution of the recursive equation P=a;P is a subset of U. Similarly, the solution of Q=b;Q is a subset of V. Therefore, a program to meet the original specification S is as follows:

P v Q v (a;(b;ZERO)) v (c;(d;ZERO)), P=a;P, Q=b;Q []

7. PROVING CHANNEL LIVENESS

Constructing a program P, in our language, to satisfy a given specification S is not hard at all; in fact, the trivial program ZERO satisfies any specification. Therefore, each constructed program should be subjected to verification to ensure that it contains a large number of sequences (provided of course that the given specification has a large number of sequences). Verification can also establish that the

constructed program keeps some channel symbols "live" in some sense. In this section, we define three (graduated) notions of channel liveness, and discuss techniques to prove that a channel symbol is live in any given program in our language.

Let S be a specification (or a program), and "a" be a channel symbol in @S. "a" is said to be *weakly live* in S iff for every positive integer n, there is a sequence s in S such that the number of occurrences of "a" in s = n.

Informally, the weak liveness of a channel symbol "a" implies that there is at least one infinite history of the system where messages are transmitted via channel "a" infinitely often. Clearly, if one channel symbol in a program P is weakly live, then P must have an infinite number of sequences. The following algorithm can decide whether a channel symbol is weakly live in a given specification.

Algorithm 1:

Input: A specification S and a channel symbol "a" in @S.

Output: A decision of whether "a" is weakly live in S.

Steps:

i. Construct the set of sequences S_a, from S, by replacing each channel symbol other than "a" by the empty sequence E.

ii. "a" is weakly live in S iff S_a can be defined by the regular expression a^*. []

Lemma 10: Algorithm 1 is correct. []

Let S be a specification, and s be a sequence in S. The sequence s is said to be *extendable* iff for any integer n greater than the length of s, S has a sequence s', of length n, such that s is a prefix of s'.

Let S be a specification, and "a" be a channel symbol in @S. "a" is said to be *live in* S iff for any extendable sequence s in S, there is a sequence s' in S such that s is a prefix of s' and s' has one more occurrence of "a" than s.

Informally, the liveness of a channel symbol "a" implies that for any infinite "history" of the system, and at any instant along this history, channel "a" can made active by transmitting a message over it. This is similar to the notion of transition liveness in Petri-Nets [9]. The following algorithm can decide whether a channel symbol is live in any specification defined by a regular expression.

Algorithm 2:

Input: A specification S, defined by a regular expression, and a channel symbol "a" in @S.

Output: A decision of whether "a" is live in S.

Steps:

i. Since S is defined by a regular expression, construct a finite automaton M which accepts each sequence in S[6]. (Each accepting state should be reachable from the initial state in M.)

ii. Represent M by a directed labelled graph G as follows: Each node in G corresponds to an accepting state in M. Each directed edge in G corresponds to a transition between two accepting states in M. Each edge in G is labelled with the alphabet symbol of its

corresponding transition in M.

iii."a" is live in S iff there is a directed path from every node, in a directed cycle, to an edge labelled "a" in G. []

Lemma 11: Algorithm 2 is correct. []

Let S be a specification. A *chain* $C = \{s_0,s_1,s_2,...\}$ *in* S is a subset of S such that s_0=E (The empty sequence), and for i=0,1,2,..., $s_{i+1}=s_i.a_i$ for some channel symbol a_i in @S. Notice that a chain is itself a specification.

Let S be a specification, and "a" be a channel symbol in @S. "a" is said to be *strongly live* iff "a" is live in each chain in S.

Algorithm 3:

Input: A specification S, defined by a regular expression, and a channel symbol "a" in @S.

Output: A decision of whether "a" is strongly live in S.

Steps:

i.Since S is defined by a regular expression, construct a finite automaton M which accepts each sequence in S. (Each accepting state should be reachable from the initial state in M.)

ii.Represent M by a directed labelled graph G as defined in step ii of Algorithm 2.

iii."a" is strongly live in S iff every directed cycle in G has at least one edge labelled "a". []

Lemma 12: Algorithm 3 is correct. []

Let S be a specification, and "a" be a channel symbol in @S. It is straightforward to show the following:

i.If "a" is strongly live in S, then "a" is live in S.

ii.If each sequence in S is extendable, and if "a" is live in S, then "a" is weakly live in S.

The above three algorithms can be used to verify that a channel symbol is weakly live, live, or strongly live in any given program in our language. Consider the program E(P,...,Q), P=F(P), ... , Q=G(Q). First, use Lemma 6 to solve each of the recursive equations in the program; these solutions can be written as follows: $P=R_p$, ... , $Q=R_q$ where R_p, ... , R_q are regular expressions. Second, substitute the program symbols P,...,Q by the regular expressions R_p, ... , R_q respectively in E(P,...,Q). The result is a regular expression R_e over the channel symbols of the program. Now, Algorithm 1, 2, or 3 can be used to show that any channel symbol in R_e is weakly live, live, or strongly live (respectively). Next we apply this methodology to show that some channel symbols in the three examples of Section 4 are strongly live.

Example (A Finite Buffer): It is required to show that the channel symbol $flow_{N+1}$, in the first example of Section 4, is strongly live. The recursive equations in this example have the following solutions:

SOURCE = $(flow_1)^*$

$CELL_i = (flow_i flow_{i+1})^* + flow_i(flow_{i+1} flow_i)^*$

$SINK = (flow_{N+1})^*$

Therefore,

$SOURCE \& CELL_1 = (flow_1 flow_2)^* + flow_1(flow_2 flow_1)^*$

Let $(SOURCE \& CELL_1)(E/flow_1)$ denotes the same expression as $SOURCE \& CELL_1$ after substituting each occurrence of "$flow_1$" by the empty sequence E, i.e.,

$(SOURCE \& CELL_1)(E/flow_1) = (flow_2)^*$

It is straightforward to show that if R_1 is any regular expression which does not contain "$flow_1$", and if "$flow_{N+1}$" is strongly live in $(SOURCE \& CELL_1)(E/flow_1) \& R_1$ then "$flow_{N+1}$" is strongly live in $(SOURCE \& CELL_1) \& R_1$.

Similarly, $((SOURCE \& CELL_1)(E/flow_1) \& CELL_2)(E/flow_2) = (flow_3)^*$. And, if R_2 is any regular expression which does not contain "$flow_2$", and if "$flow_{N+1}$" is strongly live in $((SOURCE \& CELL_1)(E/flow_1) \& CELL_2)(E/flow_2) \& R_2$, then "$flow_{N+1}$" is strongly live in $SOURCE \& CELL_1 \& CELL_2 \& R_2$.

Let T denote the expression:

$(...((SOURCE \& CELL_1)(E/flow_1) \& CELL_2)(E/flow_2)... \& CELL_N)(E/flow_N)$.

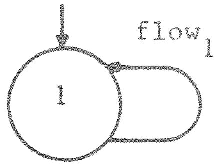
Then, by induction $T = (flow_{N+1})^*$. Also by induction, if R_N is any regular expression which does not contain "$flow_N$" and if "$flow_{N+1}$" is strongly live in $T \& R_N$ then "$flow_{N+1}$" is strongly live in $SOURCE \& CELL_1 \& ... \& CELL_N \& R_N$.

If we choose R_N to be $SINK = (flow_{N+1})^*$, then $T \& R_N = (flow_{N+1})^*$ and indeed $flow_{N+1}$ is strongly live in T & SINK, and so in $SOURCE \& CELL_1 \& ... \& CELL_N \& SINK$. This completes the required proof.

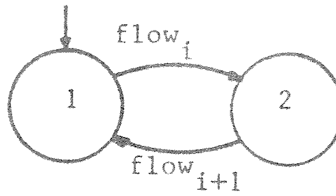
It is more convenient to follow this same proof using the finite automata for SOURCE, $CELL_i$, and SINK shown in Figures 2a, 2b, and 2c respectively. (for convenience, only accepting states and transitions between them are shown.) We first construct the finite automaton for $SOURCE \& CELL_1$, then for $SOURCE \& CELL_1 \& CELL_2$, and so on. This continues until we reach the finite automaton for $SOURCE \& CELL_1 \& ... \& CELL_N \& SINK$ and show that each directed cycle in it has at least one edge labelled "$flow_{N+1}$" (Algorithm 3).

The finite automaton for $SOURCE \& CELL_1$ is shown in Figure 2d. Since no directed cycle in this automaton consists solely of directed edges labelled "$flow_1$", and since "$flow_1$" does not appear in any of the remaining automata, then we can "hide" any edge labelled "$flow_1$" in this automaton as shown in Figure 2e. Similarly, the automaton for $SOURCE \& CELL_1 \& CELL_2$ after hiding both "$flow_1$" and "$flow_2$" is shown in Figure 2f. Therefore, the automaton for $SOURCE \& CELL_1 \& ... \& CELL_N$ after hiding "$flow_1$", "$flow_2$", ... , and "$flow_N$" is shown in Figure 2g. Finally, the automaton for $SOURCE \& CELL_1 \& ... \& CELL_N \& SINK$ is shown in Figure 2h and indeed each directed cycle in it has at least one

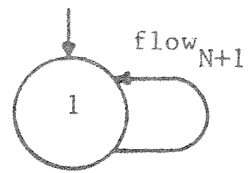
Initial State



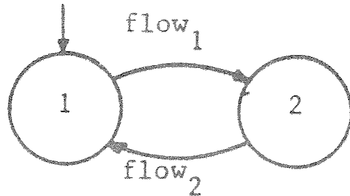
(a) SOURCE



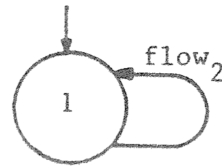
(b) $CELL_i$ (i=1..N)



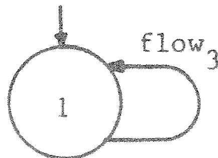
(c) SINK



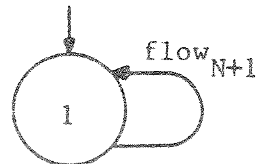
(d) SOURCE & $CELL_1$



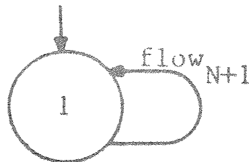
(e) SOURCE & $CELL_1$ after hiding "$flow_1$"



(f) SOURCE & $CELL_1$ & $CELL_2$ after hiding "$flow_1$" and "$flow_2$"



(g) SOURCE & $CELL_1$ &..& $CELL_N$ after hiding "$flow_1$",..., "$flow_N$"



(h) SOURCE & $CELL_1$ &...& $CELL_N$ & SINK after hiding "$flow_1$",..., "$flow_N$"

Figure 2. Finite automata to prove the strong liveness of "$flow_{N+1}$" in the finite buffer example

edge labelled "$flow_{N+1}$". []

Example (A Factorial Multiplier): It is required to show that the two channel symbols "in_1" and "out_1" are strongly live in the factorial multiplier example. Figures 3a and 3b show the finite automata for P_i (i=1,...,N-1), and P_N respectively. (As before, only accepting states and the transitions between them are shown.) The finite automaton for P_N & P_{N-1} is shown in Figure 3c. Since no directed cycle in this automaton is based solely on edges labelled "in_N" and "out_N", and since none of the remaining automata has "in_N" or "out_N", then we can hide both "in_N" and "out_N" yielding the automaton in Figure 3d. By induction, the finite automaton for P_N & P_{N-1} & ... & P_1 after hiding the channel symbols "in_i" and "out_i", i=2,...,N, is shown in Figure 3e. Since each directed cycle in this automaton has one edge labelled "i_1" and one edge labelled "out_1", both "in_1" and "out_1" are strongly live. []

Example (A Circulating Token): Consider the circulating token example discussed earlier; and assume, for convenience, that N=3. It is required to show that the three channel symbols "u_1", "u_2", and "u_3" are strongly live in this case. Figures 4a, 4b, and 4c show the finite automata for P_1, P_2, and P_3 respectively. The finite automaton of P_1 & P_2 after hiding t_2 is shown in Figure 4d; and the finite automaton of P_1 & P_2 & P_3 after hiding t_1, t_2, and t_3 is shown in Figure 4d. Since each directed cycle in this automaton has one edge labelled u_1, one edge labelled u_2, and one edge labelled u_3, then the channel symbols u_1, u_2, and u_3 are strongly live. []

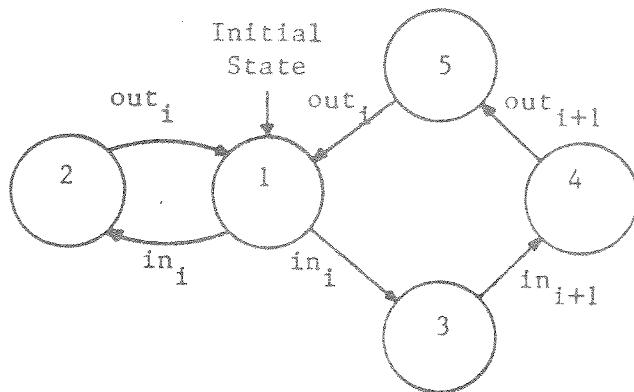
8. CONCLUDING REMARKS

We have outlined a framework to specify, program, and verify CSP systems, based on the concept of channel observation sequences. Some advantages of this approach are as follows:

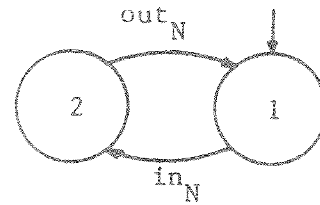
i. *It supports abstraction*: In this framework, a CSP system is defined only by the order of communications across its different channels. This is an abstract view that hides the internal operations in the system's processes except for their effect on the order of communications. Such an abstract view encourges one to think and reason about the system as a whole before worrying about its individual processes.

ii. *It highlights commonalities between similar systems*: Consider our definition of the factorial multiplier system. This definition does not really indicate any factorial multiplication function since the internal operations and so the function of each process are not defined. It is not hard to imagine a different function for each process in this system, such that the total system computes a function completely different from factorial multiplication. In other words, each system definition in our framework does not define a unique physical system but a large *class* of similar physical systems (possibly performing different functions). Highlighting these similarities between seemingly different systems is useful since it allows one to reason about a whole class of systems at the same time, instead of reasoning about these systems one by one.

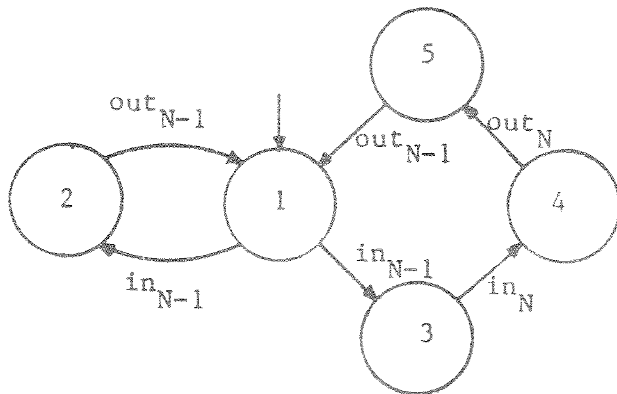
iii. *It simplifies verification of channel liveness*: As demonstrated by the examples of this paper, proving channel liveness for CSP system is relatively simple in this framework. The achieved simplicity is due to the abstraction imposed on the system definition. In many instances, the abstraction hides many of the details that do not contribute to channel liveness (e.g., internal operations) while preserving those that do, namely the different orders of communications.



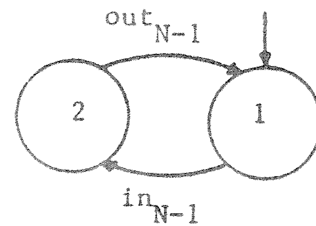
(a) P_i $(i=1,\ldots,N-1)$



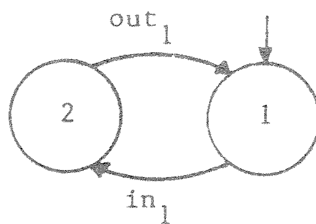
(b) P_N



(c) P_N & P_{N-1}



(d) P_N & P_{N-1} after hiding "in$_N$" and "out$_N$"



(e) P_N & P_{N-1} & ... & P_1 after hiding "in$_i$" and "out$_i$", $i=2,\ldots,N$

Figure 3. Finite automata to prove strong liveness for "in$_1$" and "out$_1$" in the factorial multiplier example

Initial State

(a) P_1

(b) P_2

(c) P_3

(d) P_1 & P_2 after hiding "t_2"

(e) P_1 & P_2 & P_3 after hiding "t_1", "t_2" and "t_3"

Figure 4. Finite automata to prove strong liveness for u_1, u_2 and u_3 in the circulating token example (N=3)

The outlined framework can be extended in two directions:

i. *Extend the expressive power of the language*: As discussed in Sectionn 4, it seems useful to extend our programming language to allow general recursion. Also, it seems useful to allow processes to exchange data values instead of mere noninterpreted messages. These extensions should be introduced without disturbing the algebraic properties, discussed in Section 5, of our current language.

ii. *Extend the programming and verification support for the language*: The current programming and verification methodologies should be extennded to support the language extensions suggested in i. In particular, the extended verification methodology should support the verification of safety as well as livenness properties.

REFERENCES

1. M. G. Gouda and W. Chu, "A finite state model for protocol processes and services," Technical Report-198, Department of Computer Sciences, University of Texas at Austin, April 1982. Submitted for publication.
2. E. C. R. Hehner and C. A. R. Hoare, "Another look at communicating processes," Technical Report CSRG-134, Computer Systems Research Group, University of Toronto, July 1981.
3. C. A. R. Hoare, "Communicating sequential processes," Comm. of the ACM, Vol. 21, No. 8, August 1978, pp666-677.
4. C. A. R. Hoare, "A calculus of total correctness for communicating processes," Science of Computer Programming, Vol. 1, No. 1, 1982.
5. C. A. R. Hoare, "Specifications, programs, and implementations," Technical Monograph PRG-29, Programming Research Group, Oxford University, June 1982.
6. J. E. Hopcroft and J. D. Ullman, Introduction to automata theory, languages and computation, Addison-Wesley, Reading, Mass., 1979.
7. P. Merlin and G. V. Bochmann, "On the construction of submodule specifications and communication protocols," TOPLAS, Vol. 5, No. 1, Jan. 1983.
8. R. Milner, A calculus of communicating systems. Lecture Notes on Computer Science 92, 1980.
9. J. L. Peterson, Petri net theory and the modeling of systems, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1981.

APPENDIX:PROOFS OF LEMMAS

Lemma 1:

Let $s_1 = \#_{@P}s$ and $s_2 = \#_{@Q}s$

s_1 is in P, s_2 is in Q, and $\#_{@P\cup @Q}s = s$

iff s_1 and s_2 are compatible with respect to $@P\cap @Q$

iff s_1 and s_2 can be written as:

$s_1 = x_0.a_1.x_1. \ldots .a_k.x_k$, and

$s_2 = y_0.a_1.y_1. \ldots .a_k.y_k$

where $a_1,\ldots,a_k$ are symbols in $@P\cap @Q$

$x_0,\ldots,x_k$ are strings over symbols in @P-@Q, and

$y_0,\ldots,y_k$ are strings over symbols in @Q-@P.

iff s can be written as:

$s = i(x_0,y_0).a_1.i(x_1,y_1). \ldots .a_k.i(x_k,y_k)$

where $a_1,\ldots,a_k$ are symbols in $@P\cap @Q$, and

for $j=0,\ldots,k$ $i(x_j,y_j)$ is a string in $intrlv(x_j,y_j)$.

iff s in P&Q. []

Lemma 2: $a;(P\vee Q) = E \cup \{a.s \mid s \text{ is in } P \cup Q\}$

$= E \cup \{a.s \mid s \text{ is in } P\} \cup \{a.s \mid s \text{ is in } Q\}$

$= (a;P) \vee (a;Q)$ []

Lemma 3: The operator v is the union operator; and so it is idempotent, commutative, associative, and distributes through v. []

Lemma 4:

i. P & P = P:

s is in P&P

iff $\#_{@P}s$ is in P and $\#_{@P}\, s = s$ <Lemma 1>

iff s is in P

ii. P & Q = Q & P:

s is in P&Q

iff $\#_{@P}s$ is in P, $\#_{@Q}s$ is in Q, and $\#_{@P\cup @Q}s = s$ <Lemma 1>

iff s is in Q&P

iii. (P&Q)&R = P &(Q&R):

s is in (P&Q)&R

iff $s_1 = \#_{@P\cup @Q}s$ is in P&Q, $\#_{@R}s$ is in R, and $\#_{@P\cup @Q\cup @R}s = s$

iff $\#_{@P}s_1$ is in P, $\#_{@Q}s_1$ is in Q, $\#_{@R}s$ is in R, and $\#_{@P\cup @Q\cup @R}s = s$

iff $\#_{@P}s$ is in P, $\#_{@Q}s$ is in Q, $\#_{@R}s$ is in R, and $\#_{@P\cup @Q\cup @R}s = s$

iff $\#_{@P}s$ is in P, $\#_{@Q\cup @R}s$ is in Q&R, and $\#_{@P\cup @Q\cup @R}s = s$

iff s is in P &(Q & R)

iv. P&(QvR) = (P&Q)v(P&R):

s is in P&(QvR)

iff $\#_{@P}s$ is in P, $\#_{@QU@R}s$ is in QvR, and $\#_{@PU@QU@R}s = s$

iff $\#_{@P}s$ is in P and [$\#_{@QU@R}s$ is in Q or $\#_{@QU@R}s$ is in R] and $\#_{@PU@QU@R}s = s$

iff $\#_{@P}s$ is in P and [($\#_{@Q}s$ is in Q and $\#_{@PU@Q}s = s$) or ($\#_{@R}s$ is in R and $\#_{@PU@R}s = s$)] and $\#_{@PU@QU@R}s = s$

iff s is in (P&Q)v(P&R) []

Lemma 5: An expression E(P,...,Q) in our language is composed of:

i. the program symbols P,...,Q (each of them denotes a set of strings over the channel symbols in A),

ii. the constants ZERO and CHAOS (each of them denotes a set of strings over the channel symbols in A), and

iii. the operators "a;", "v", "&" (each of them, when applied to some set(s) of strings over the channel symbols in A, will produce a set of strings over the channel symbols in A).

Therefore E(P,...,Q) defines a set of strings over the channel symbols in A. []

Lemma 6: First, we prove two propositions which are used later to prove Lemma 6.

Proposition 1: Let F(P) be an expression, in our language, that is based on some program symbol P. Then,

$$F(\lim n\to\infty.F^n(A^*)) = \lim n\to\infty.F^n(A^*)$$

where A is the channel alphabet.

Proof: It is straightforward to show (by induction on n) that for n=0,1,2..., $F^{n+1}(A^*)$ is a subset of $F^n(A^*)$ where $F^0(A^*)=A^*$, and $F^{n+1}(A^*)=F(F^n(A^*))$. Therefore, for any channel symbol "a", and for any set Q of strings over the channel alphabet A, we have:

$$a;(\lim n\to\infty.F^n(A^*)) = \lim n\to\infty.a;F^n(A^*),$$
$$Qv(\lim n\to\infty.F^n(A^*)) = \lim n\to\infty.QvF^n(A^*), \text{ and}$$
$$Q\&(\lim n\to\infty.F^n(A^*)) = \lim n\to\infty.Q\&F^n(A^*).$$

This leads to:

$$\begin{aligned}F(\lim n\to\infty.F^n(A^*)) &= \lim n\to\infty.F(F^n(A^*))\\ &= \lim n\to\infty.F^{n+1}(A^*)\\ &= \lim n\to\infty.F^n(A^*).\end{aligned}$$

-- end of proposition 1 --

Proposition 2: Let F(P) be an expression, in our language, that is based on some program symbol P; and let Q and R be two sets of strings over the channel alphabet A. If Q⇒R (i.e., Q is a subset of R), then F(Q)⇒F(R). (F is *monotonic.*)

Proof: Since Q⇒R, then there exists a set S of strings over the channel alphabet A such that R=QvS.

(i) Since a;R = a;(QvS)
= (a;Q)v(a;S),
then a;Q ⇒ a;R

(ii) Since TvR = Tv(QvS)
= (TvQ)v(TvS),
then TvQ ⇒ TvR

(iii) Since $T\&R = T\&(QvS)$
$= (T\&Q)v(T\&S)$,
then $T\&Q \Rightarrow T\&R$

From (i), (ii) and (iii), and by induction on the number of operators in F, we get: $F(Q)\Rightarrow F(R)$.

-- end of proposition 2 --

Now Lemma 6 can be proven in two steps. First we show that $\lim n\rightarrow\infty.F^n(A^*)$ satisfies the equation; i.e., it is a potential solution. Second, we show that it is the weakest solution; i.e., for any set S which satisfies the equation, S is a subset of $\lim n\rightarrow\infty.F^n(A^*)$, denoted $S\Rightarrow\lim n\rightarrow\infty.F^n(A^*)$.

To show that $\lim n\rightarrow\infty.F^n(A^*)$ satisfies $P=F(P)$, notice that $F(\lim n\rightarrow\infty F^n(A^*))=\lim n\rightarrow\infty.F^n(A^*)$ by proposition 1.

Let S be any set of strings over the channel alphabet A; and assume that S satisfies $P=F(P)$. Then $S=F(S)=F^2(S)=...=F^n(S)=...=\lim n\rightarrow\infty.F^n(S)$ But since $S\Rightarrow A^*$, then $F^n(S)\Rightarrow F^n(A^*)$ by proposition 2. Therefore, $S=\lim n\rightarrow\infty.F^n(S)\Rightarrow\lim n\rightarrow\infty F^n(A^*)$. []

Lemma 7: Each set used in E(P,...,Q) is closed under prefixing; and eah operator in E(P,...,Q) preserves closure under prefixing. Therefore, the set defined by E(P,...,Q) is closed under Prefixing. []

Lemma 8: The set A^* is closed under prefixing; each constant in F is closed under prefixing; and each operator in F preserves closure under prefixing. Therefore, for any n, $F^n(A^*)$ is closed under prefixing; and $\lim n\rightarrow\infty.F^n(A^*)$ is closed under prefixing. []

Lemma 9:

1. **Correctness of the First Step:** we need to show that if $E(U,...,V)\Rightarrow S$, $P\Rightarrow U$,..., and $Q\Rightarrow V$, then $E(P,...,Q)\Rightarrow S$. The proof follows from Proposition 2 in the proof of Lemma 6; notice that $E(P,...,Q)\Rightarrow E(U,...,Q)\Rightarrow...\Rightarrow E(U,...,V)\Rightarrow S$.
2. **Correctness of the Second Step:** we need to show that if $F(UvS_n)\Rightarrow UvS_{n+1}$, then the solution of $P=F(P)$ is a subset of U (i.e., $\lim n\rightarrow\infty.F^n(A^*)\Rightarrow U$). For $n=0,1,...$, $F^n(A^*)=F^n(S_0)=F^n(UvS_0)\Rightarrow UvS_n$. Taking the limit as n goes to infinity, we get:
   $\lim n\rightarrow\infty.F^n(A^*) \Rightarrow \lim n\rightarrow\infty.(UvS_n)$, i.e.,
   $\lim n\rightarrow\infty.F^n(A^*) \Rightarrow Uv(\lim n\rightarrow\infty.S_n) = U$. []

Lemma 10:

$S_a = a^*$

iff for any positive integer n, there is a string s in S

such that the number of occurrences of "a" in s = n

iff "a" is weakly live in S. []

Lemma 11: Each extendable sequence in S corresponds to a node in a directed cycle in G, and vice versa. Therefore,

there is a directed path from any node, in a directed cycle,

to an edge labelled "a" in G

iff for any extendable sequence s in S, there is a sequence s'

in S such that s is a proper prefix of s', and s' has one

more occurrence of "a" than s

iff "a" is live in S. []

Lemma 12: Each chain in S corresponds to one or more directed cycles in G; and each directed cycle in G corresponds to a chain in S. Therefore,

there is an edge labelled "a" in each directed cycle in G

iff "a" is live in each chain in S

iff "a" is strongly live in S. []