Validation of systems of parallel processes

A. M. Lister

Computing Centre, University of Essex, Wivenhoe Park, Colchester, Essex, CO4 3SQ

An algorithm is presented for verifying whether systems of parallel processes behave in the way expected of them. The basis of the algorithm is the combination of state transitions of individual processes into state transitions of the complete system. The algorithm is realised in a program whose input is a simple formalised description of each process and whose output gives details of the behaviour of the system as a whole. By way of illustration the algorithm is used to validate a solution to the readers and writers problem.

(Received February 1973)

1. Introduction

A good deal of attention has recently been paid, (e.g. Hansen, 1972; Habermann, 1972; Gilbert and Chandler, 1972), to the proof of correctness of systems of parallel processes. The focussing of attention has been caused by the aesthetic appeal of the problem and by its practical application in the construction of logically correct operating systems.

Habermann's work is confined to systems in which processes are synchronised by semaphores (Dijkstra, 1965). He establishes a relation which is invariant under *P*- or *V*-operations on semaphores, and uses it to deduce properties of the system under consideration. This approach is attractive, but is limited by the fact that a separate proof must be constructed for every system and by the difficulty of applying the technique to large systems.

Gilbert and Chandler's approach is to infer the behaviour of the system as a whole from the specification of each of its constituent processes. The technique is to formulate state transition rules for each process, and then to combine these rules to obtain transition rules for the complete system. The attractiveness of the method lies in the possibility of automating the procedures by which transition rules are combined and by which the composite states of the complete system are examined. However, limitations arise from the way in which the state transition rules must be specified: explicit rules must be given for all transformations which alter the value of a state variable, so that variables with a potentially infinite set of values (e.g. semaphores, whose values can be arbitrarily large) require an infinite set of transition rules. In practice one 'knows' that variables will take on no more than a small subset of their possible values, but this is to assume the very properties of the system which one is trying to prove.

The algorithm presented in this paper is a generalisation of Gilbert and Chandler's technique which overcomes these deficiencies. Individual state transitions are expressed implicitly, so that the number of transition rules is always finite and generally small. This ensures that the computation and analysis of the composite states of the entire system can be performed in an acceptable period of time. Details of program size and computation time for the analysis of a typical system are given in Section 4.

2. Basis of the algorithm

The algorithm is based on the combination of state transition rules for individual processes, specified in isolation, into state transition rules for the system of processes as a whole. The two important features of the algorithm are

1. The way in which the individual transition rules are specified.
2. The way in which the transition rules are combined.

The first of these features is described in Section 2.1, the second in Section 2.2.

2.1. *Partial transformations and partial rules*

We postulate a system consisting of a number of processes $P_1, P_2, \ldots, P_n$ which act on a set of data variables $W_1, W_2, \ldots, W_k$. A particular set of values of the data variables is represented by a k-tuple $\mathbf{w} = (w_1, w_2, \ldots, w_k)$, and $W = \{\mathbf{w}\}$ is the set of all such possible k-tuples.

The points in the progress of a process P_i between which it tests or modifies the value of a data variable are categorised as *states* of P_i. Thus a process moves from one state to another by testing or modifying the value of a variable (e.g. by operating on a semaphore). We assume a finite number of states for each process, and denote the state set of P_i by $S_i = \{s_{i1}, s_{i2}, \ldots, s_{in_i}\}$.

The transition of P_i from one state to the next is specified by defining a *partial transformation*

$$f_i : S_i \times W \rightharpoonup S_i .$$

That is, a partial transformation is a partial function of the current state and the data variables which yields the next state of the process. The partiality of the function is due to the fact that a transition out of a particular state will in general be possible only when the data variables have certain values.

A 'side-effect' of a partial transformation is the possible alteration in the value of some of the variables. If, for example, one of the variables is a semaphore, and a partial transformation depends on a *P*-operation on it, then the side effect is a possible decrement of its value. Hence we say that the partial transformation f_i is accompanied by the *variable transformation*

$$g_i : S_i \times W \rightharpoonup W .$$

Thus g_i shows the effect on the data variables corresponding to a change of state of process P_i.

A change of state of P_i, together with the corresponding changes in the values of the data variables, is completely specified by combining the appropriate partial transformation with the accompanying variable transformation in a *partial transition*

$$T_i : S_i \times W \rightharpoonup S_i \times W$$

where

$$T_i(s_i, \mathbf{w}) = (f_i(s_i, \mathbf{w}), g_i(s_i, \mathbf{w})) .$$

The set of all $(s_i, \mathbf{w})$ for which T_i is defined gives a set of *partial rules* for P_i of the form

$$(s_i, \mathbf{w}) \rightarrow (f_i(s_i, \mathbf{w}), g_i(s_i, \mathbf{w})) .$$

Each partial rule is an instance of a partial transition.

Since each partial rule operates on a state of only one process, the set of partial rules for each process can be written down in isolation, with no regard to the behaviour of any other process.

The set of partial rules is in fact a formalised program executed by the process.

For example, suppose the system contains a cyclic process Q which passes through a critical section (Dijkstra, 1965) protected by a mutual exclusion semaphore *sem*. Then the program obeyed by Q may be written

L: $P(sem)$;
critical section
$V(sem)$;
goto L;

and we may conveniently identify two states of Q: namely, state 0 when Q is outside the critical section, and state 1 when it is inside the critical section. The partial transformation on state 0 is a function conditional on the value of *sem*, and the corresponding variable transformation is a possible decrement of *sem*. The partial transformation on state 1 is an unconditional transition to state 0, with associated variable transformation an increment of *sem*.

Hence the partial rules for Q, assuming that *sem* is the only data variable, are

$$(0)(sem) \rightarrow (f_Q(0, sem))(g_Q(0, sem))$$
$$(1)(sem) \rightarrow (f_Q(1, sem))(g_Q(1, sem))$$

where

$$f_Q(0, sem) = \textbf{if sem} = 0 \textbf{ then } 0 \textbf{ else } 1$$
$$f_Q(1, sem) = 0$$
$$g_Q(0, sem) = \textbf{if } sem = 0 \textbf{ then } 0 \textbf{ else } sem - 1$$
$$g_Q(1, sem) = sem + 1$$

Functions similar to f_Q and g_Q. which are derived from operations on semaphores, occur so frequently in practice that they are accorded special names and built into the programmed form of the algorithm as described in Section 3.

2.2. *Composite states and composite transformations*

Given the set of processes $\{P_i\}$ each with state set S_i, it is possible to define a state set $S = S_1 \times S_2 \times \ldots \times S_n$ for the set of processes as a whole. The *composite state* at any instant of the entire system of processes and data variables is an element of the set $S \times W$; that is, it is of the form

$$(s_1, s_2, \ldots, s_n)(w_1, w_2, \ldots, w_k) .$$

The composite state may be changed by the change of state of any individual process together with any corresponding change to the data variables. Specifically, the composite state is changed according to the relation

$$T: S \times W \rightarrow S \times W$$

defined by

$$(\mathbf{s}, \mathbf{w}) \overset{T}{\rightarrow} (\mathbf{s}', \mathbf{w}') \text{ if, and only if, } \exists i (1 \leqslant i \leqslant n)$$

such that

(1) $\forall_j \neq i, s_j' = s_j$
(2) $s_i' = f_i(s_i, \mathbf{w})$
(3) $\mathbf{w}' = g_i(s_i, \mathbf{w})$.

Condition (1) implies that only one process P_i may change state at a time; conditions (2) and (3) imply that the change of state of P_i and the corresponding changes to the data variables are produced by the application of one of the partial rules for P_i.

It will be noted that condition (1) disallows the possibility of several processes changing state at the same instant. The relaxation of this restriction is discussed in Section 5. It can however, be remarked that the condition is not unrepresentative of many real-life situations; in particular it applies when:

(*a*) all processes share a single processor, or
(*b*) processes running on several processors communicate via variables (e.g. semaphores or status bits) to which access is restricted by hardware to a single process at a time.

A change of composite state $(\mathbf{s}, \mathbf{w}) \overset{T}{\rightarrow} (\mathbf{s}', \mathbf{w}')$ is called a *composite transformation*. The non-determinacy of the system is reflected in the fact that T is a relation rather than a function, so that a composite state may have more than one successor under T.

Given an initial composite state it is possible to apply systematically the partial rules of all processes to generate a *state graph* whose arcs are the possible composite transformations and whose nodes are the attainable composite states. Properties of the system are deduced by analysing the state graph; in particular:

(*a*) the finiteness of the graph is a necessary condition for the system to be well formed.
(*b*) the existence of 'cul-de-sac' nodes indicates the existence of potential deadlock states.
(*c*) the attainability of any composite state is indicated by the existence of the corresponding node.

3. Implementation

A successful implementation of the algorithm depends largely on the ease with which composite states and partial rules can be expressed. To facilitate expression, standard names, listed below, are assigned to commonly occurring partial and variable transformations. Variable values or states which are not relevant to a partial rule are denoted by x. (Thus x, occurring as a variable value means 'any non-specified value', and as a state means 'any non-specified state'). Variable transformations are split into their component parts (one component per variable), and all null components are denoted by N. Implicit parameters of partial transformations, or of components of variable transformations, (i.e. the state or variable being transformed), are omitted from statements of partial rules. Other parameters are written as subscripts to the transformation names (see the example below). States of individual processes are generally numbered 0, 1, 2, ...

The standard transformation names are:

1. PV: a variable transformation corresponding to a P-operation on a semaphore (this is the first instance of g_Q in the example of Section 2.1).
2. PS: a partial transformation of a state dependent on a P-operation on a semaphore (the first instance of f_Q in the example). Parameters are the index number of the semaphore in the set of variables and the transformed state if the P-operation is successful.
3. V: a variable transformation corresponding to a V-operation on a semaphore (the second instance of g_Q in the example).
4. TV: a variable transformation corresponding to a test on the value of the variable. Parameters are the value against which to test for equality, the new value if the test succeeds, and the new value if the test fails.
5. TS: a partial transformation dependent on the testing of the value of some variable. Parameters are as for TV, plus the index number of the variable to be tested.
6. D: a transformation which decrements the state number or variable to which it is applied.
7. I: as D, but increment instead of decrement.
8. N: a null transformation (identity function).

To illustrate this notation we take the example of two cyclic processes P_1 and P_2 each passing through a critical section protected by a single semaphore. The code obeyed by P_1 and P_2, and the states entered, are the same as for process Q in Section 2.1. The partial rules for each process are identical, viz.:

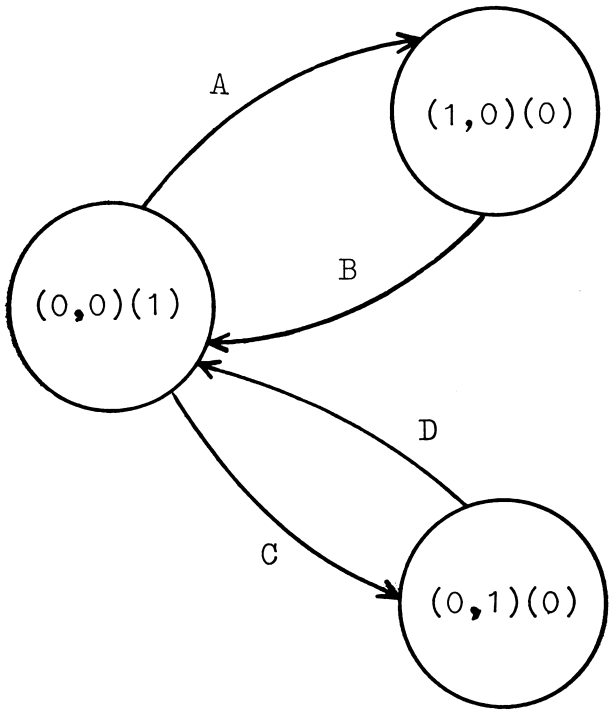

Fig. 1 **A: composite transformation from rule (1) of P_1**
B: composite transformation from rule (2) of P_1
C: composite transformation from rule (1) of P_2
D: composite transformation from rule (2) of P_2

$$(0)(x) \rightarrow (PS_{1,1})(PV) \quad (1)$$
$$(1)(x) \rightarrow (0)(V) \quad (2)$$

The reader will note the comparatively simple form of these rules, which are the same as those given earlier for process Q. The subscripts of the transformation PS refer to the index number of the semaphore in the set of data variables (it is in fact the only data variable) and transformed state if the P-operation is successful.

Application of the partial rules to the initial composite state (0, 0)(1), in which both processes are outside their critical sections and the value of the semaphores is 1, yields the state graph shown in **Fig. 1.**

Analysis of the graph reassuringly shows that there are no deadlock states and that the processes are never simultaneously in their critical sections (composite state (1, 1)(x)).

4. An example of system validation

The algorithm has been implemented as a program written in POP-2 (Burstall, Collins and Popplestone, 1971), and has been used to validate several systems of parallel processes. In particular it has been used in a current research project on operating system design and to check student solutions to problems in parallel programming. The program includes as built-in functions all the commonly occurring transformations listed in Section 3 and also allows the user to define other transformations of his own.

To illustrate the action of the program we give as an example the validation of a recent solution to the readers and writers problem (Courtois, Heymans, and Parnas, 1971). The problem is that of a set of 'readers' and a set of 'writers' who are accessing a common database. Access is to be controlled so that only one writer may write at a time, and no writers may write while a reader is reading. Any number of readers may read simultaneously.

The solution to the problem, with labels inserted to indicate the delineation of states, is

Readers	*Writers*
*state*0 : $P(mutex)$;	
*state*1 : $readcount := readcount + 1$;	
*state*2 : **if** $readcount = 1$ **then**	
*state*3 : $P(w)$;	
*state*4 : $V(mutex)$;	*state*0 : $P(w)$;
*state*5 : *perform reading*	*state*1 : *perform writing*
$P(mutex)$;	$V(w)$;
*state*6 : $readcount := readcount - 1$;	
*state*7 : **if** $readcount = 0$ **then**	
*state*8 : $V(w)$;	
*state*9 : $V(mutex)$;	

The data variables are the semaphores *mutex* and *w* and the variable *readcount*. They will be referred to in that order in partial rules and composite states.

The partial rules are

Readers	*Writers*
$(0)(x, x, x) \rightarrow (PS_{1,1})(PV, N, N)$	$(0)(x, x, x) \rightarrow (PS_{2,1})(N, PV, N)$
$(1)(x, x, x) \rightarrow (2)(N, N, I)$	$(1)(x, x, x) \rightarrow (O)(N, V, N)$
$(2)(x, x, x) \rightarrow (TS_{3,1,3,4})(N, N, N)$	
$(3)(x, x, x) \rightarrow (PS_{2,4})(N, PV, N)$	
$(4)(x, x, x) \rightarrow (5)(V, N, N)$	
$(5)(x, x, x) \rightarrow (PS_{1,6})(PV, N, N)$	
$(6)(x, x, x) \rightarrow (7)(N, N, D)$	
$(7)(x, x, x) \rightarrow (TS_{3,0,8,9})(N, N, N)$	
$(8)(x, x, x) \rightarrow (9)(N, V, N)$	
$(9)(x, x, x) \rightarrow (0)(V, N, N)$	

N.B.: The subscripts of transformation TS are such that $TS_{3,1,3,4}$ means 'if variable number 3 is equal to 1 then transform to state 3, otherwise transform to state 4'.

The system is validated for two readers and two writers, on the assumption that if it works in this case it will work in all cases. Part of the console record is given in **Fig. 2,** which is largely selfexplanatory. The following remarks indicate how the validation proceeds and should be read in conjunction with Fig. 2.

```
TYPE "H" FOR HELP WITH BUILT-IN STATE TRANSFORMATIONS
OTERWISE TYPE "N". : N
DO YOU WANT TO DEFINE ANOTHER FUNCTION ? : N

HOW MANY PROCESSES ? : 4
HOW MANY DATA VARIABLES ? : 3
ESTIMATED NO. OF ATTAINABLE COMPOSITE STATES ? : 100
HOW MANY PARTIAL RULES FOR PROCESS  1 ? : 10
HOW MANY PARTIAL RULES FOR PROCESS  2 ? : 10
HOW MANY PARTIAL RULES FOR PROCESS  3 ? : 2
HOW MANY PARTIAL RULES FOR PROCESS  4 ? : 2
PARTIAL RULES FOR PROCESS 1
LHS OF RULE 1
STATE ? : 0
VARIABLES ? : X X X
STATE TRANSFORMATION ? : PS
SEM ? : 1
NEW STATE ? : 1
VARIABLE TRANSFORMATIONS
T 1 ? : PV
T 2 ? : N
T 3 ? : N
LHS OF RULE 2
STATE ? : 1
VARIABLES ? : X X X
STATE TRANSFORMATION ? : 2
VARIABLE TRANSFORMATIONS
T 1 ? : N
T 2 ? : N
T 3 ? : I
LHS OF RULE 3
STATE ? : 2
VARIABLES ? : X X X
STATE TRANSFORMATION ? : TS
WHICH VARIABLE ? : 3
```

```
WHAT VALUE ? : 1
SUCCESS STATE ? : 3
FAIL STATE ? : 4
VARIABLE TRANSFORMATIONS
T 1 ? : N
T 2 ? : N
T 3 ? : N


PARTIAL RULES FOR PROCESS 2
SAME AS A PREVIOUS PROCESS ? : Y
WHICH ? : 1
PARTIAL RULES FOR PROCESS 3
SAME AS A PREVIOUS PROCESS ? : N
LHS OF RULE 1
STATE ? : 0
VARIABLES ? : X X X
STATE TRANSFORMATION ? : PS
SEM ? : 2
NEW STATE ? : 1
VARIABLE TRANSFORMATIONS
T 1 ? : N
T 2 ? : PV
T 3 ? : N
LHS OF RULE 2
STATE ? : 1
VARIABLES ? : X X X
STATE TRANSFORMATION ? : 0
VARIABLE TRANSFORMATIONS
T 1 ? : N
T 2 ? : V
T 3 ? : N
PARTIAL RULES FOR PROCESS 4
SAME AS A PREVIOUS PROCESS ? : Y
WHICH ? : 3
SATISFIED ? : Y
INITIAL COMPOSITE STATE ? : 0 0 0 0 1 1 0

NO. OF ATTAINABLE COMPOSITE STATES IS   50
NO. OF EFFECTIVE COMPOSITE TRANSITIONS IS   88
DEADLOCK STATES
NONE

HOW MANY STATES OF INTEREST ? : 3
TYPE THEM ONE AT A TIME
: X X 1 1 X X X
NONE ATTAINED
: 5 X 1 X X X X
NONE ATTAINED
: 5 5 X X X X X
( 5 5 0 0)( 1 0 2)
PRINTING OF ALL ATTAINABLE STATES ? : N

PRINTING OF ALL EFFECTIVE TRANSITIONS ? : N


ANY MORE WITH THIS SYSTEM ? : N
END OF STATE COMPOSITION
```

Fig. 2

The estimated number of attainable composite states is used by the program as a check on the size of the state graph. If more states than this are generated the user is advised that the system may be ill-formed, and asked if he wishes to revise the estimate. The initial composite state is (0, 0, 0, 0)(1, 1, 0), i.e. no readers or writers active and both semaphores set to 1. States of interest supplied to the program are:

(i) $(x, x, 1, 1)(x, x, x)$: 2 writers active simultaneously
(ii) $(5, x, 1, x)(x, x, x)$: A reader and a writer active simultaneously
(iii) $(5, 5, x, x)(x, x, x)$: 2 readers active simultaneously

The first two of these are forbidden by the constraints of the problem, and the program confirms that no such states are attained. The third is a state we wish to occur, and it can be seen that the solution does indeed allow states of this form.

Thus the solution is validated in the sense that:

1. the number of attainable states is finite
2. there are no deadlock states
3. forbidden states are not attained, whereas desired states are attained.

It should be noted that the decomposition into states given above, which was designed for maximum clarity, is not the only one possible. For example, state 1 for readers can be eliminated by defining a new transformation from state 0 to state 2 which combines the separate transformations *PS* and *I*. This has the effect of reducing the number of partial rules and composite states and hence reducing the computation time required for the validation.

The program, running on a PDP-10 with a mean store access time of 1.1 microseconds, takes 8 seconds to compile and 21 seconds to perform the validation shown in Fig. 2. The size of the program is 3K, plus 10K for the POP-2 system, and the data area for this run is 1.3K.

5. Conclusion

The author feels that the algorithm described in this paper can be a useful tool in the construction of logically correct operating systems and real-time control programs. A crucial factor which could limit its usefulness is the rate at which computation time increases with the number of processes and variables. Current evidence suggests that the transformation functions in practice are so partial that the number of states and the complexity of the state graph are kept within reasonable bounds. Further research is being undertaken in this area and will be reported in a subsequent paper.

The restriction that only one process may change state at a time is, as mentioned in Section 2.2, representative of many real-life situations. The restriction may be successively relaxed by allowing simultaneous changes of state which:

(i) involve only disjoint subsets of the variables.
(ii) have the same effect on common variables.
(iii) have different effects on common variables.

The last of these relaxations corresponds to the existence of 'race conditions' between processes, and lends another degree of indeterminacy, and hence a larger number of successor states to the system. The effects of making the relaxations are currently being investigated.

It may be remarked that depending upon how the 'state' of a process is defined, the algorithm may be used to validate systems at any level of design; invocation of parts of the system which have been validated at a low level can be regarded as single state transitions at a higher level.

The algorithm should also be applicable to other fields, such as logic design, where state composition is a major activity.

6. Acknowledgements

I am grateful to my colleagues Dr. J. M. Brady and Dr. J. G. Laski for their helpful comments on a first draft of this paper.

References

BURSTALL, R. M., COLLINS, J. S., and POPPLESTONE, R. J. (1971). *Programming in POP-2*, Edinburgh University Press.

COURTOIS, P. J., HEYMANS, R., and PARNAS, D. L. (1971). Concurrent Control with 'Readers' and 'Writers'. *CACM.*, Vol. 14, No. 10, pp. 667-668.

DIJKSTRA, E. W. (1965). Co-operating Sequential Processes. *Report EWD* 123, *Technological University of Eindhoven.* Also in *Programming Languages*, F. Genuys, Ed., Academic Press (1968), pp. 43-112.

HABERMANN, A. N. (1972). Synchronisation of Communicating Processes. *CACM.*, Vol. 15, No. 3, pp. 171-176.

HANSEN, P. B. (1972). A Comparison of Two Synchronising Concepts. *Acta Information*, Vol. 1, No. 3, pp. 190-199.

GILBERT, P. and CHANDLER, W. J. (1972). Interference Between Communicating Parallel Processes. *CACM*, Vol. 15, No. 6, pp. 427-437.