Verifying Properties of Large Sets of Processes with Network Invariants (Extended Abstract)

Pierre Wolper Vinciane Lovinfosse
Université de Liège
Institut Montéfiore, B28,
4000 Liège Sart-Tilman, Belgium

1 Introduction

If a system is built from a large number of identical finite-state processes, it seems intuitively obvious that, with the help of "a little induction", the verification of such a system can be reduced to a finite-state problem. The difficulty is to find the right form of "a little induction". There have been several attempts to address this problem in the context of model-checking [CGB86], [CG87], [GS87]. In very general terms (see Section 6 for more details), the approach is to find ways of proving that if a process satisfies a formula, then the n-fold parallel composition of this process with itself still satisfies the same (or a related) formula. This approach makes some interesting verifications possible. However, it has its limits and usually requires the implementation of special purpose tools.

In this paper, we propose an alternative approach. It is an attempt to make the "little induction" explicit and simple. If one wants to prove that some property holds for the composition of n processes P, one ought to be able to proceed as follows. Prove that one process satisfies the property or, as is often necessary when using induction, a stronger property I. Then prove that the composition of any process satisfying I with one of the processes P still satisfies I. Such a property I essentially represents the joint behavior of any number of processes P. Since adding one more process P to a network satisfying I does not change I, we call it a *network invariant.* All this is general and quite obvious. The problem is to find a framework in which it works.

For this, we turn to process theory in the style of CCS and CSP [Mil80], [Hoa85]. We actually use a variant of TCSP, but this choice is not important as long as some conditions made explicit in Section 2 are satisfied. The idea is that the network invariant I is itself expressed as a process. The inductive step then essentially reduces to proving in the process theory that $I \parallel P$ is a process equal to or stronger than I. Of course if the processes are finite-state, this can be done with an automatic verification tool. Hence, once the invariant I is found, our method is completely automatic.

Clearly, the crucial point is finding the invariant or more fundamentally determining if such an invariant indeed exists. We prove that invariants do not always exist and that determining if an invariant does exist is undecidable. Nevertheless, we have been able to handle some very interesting examples. Among these, we can cite a version of the distributed mutual exclusion problem studied in [CGB86] and a buffer composed of identical elementary (size 1) buffers. Note that for the buffer example, the invariant should essentially represent the behavior of a buffer of arbitrary size. This might appear to be impossible in the context of finite-state systems. However, this problem can be solved with the help of the concept of "data-independence" described in [Wol86]. There, it is shown that if a process has a behavior independent of the value of the data it is manipulating, it is possible to specify within a finite-state framework that it behaves like a buffer.

One might argue that having to find the invariant is a drawback of our method since this step is not algorithmic. We can answer, that the invariant is often close to the specification to be proved and that there is no miracle. Proving interesting properties by induction sometimes requires a little imagination. This is true of inductive proofs in all contexts. Moreover, once the invariant has been found, our verification method reduces to a traditional finite-state verification problem which can be solved with existing tools. No specific implementation is thus necessary to use the method. Finally, let us note that our method can handle any type of property, safety or liveness. For the latter type of property though, it can be necessary to turn to an extension of process theory towards infinite behaviors as studied in [Par85] and [ALW89]. This is straightforward as our method is compatible with essentially any process theory.

In the next section, we give a general description of our method. We then particularize it to a variant of CSP. Thereafter, we show how data-independence can be used to find suitable invariants. Next we deal with examples and finally we conclude and compare our work to existing results.

2 General Framework

Consider a set of processes **P** and a a relation $\leq$ on this set such that $P_1 \leq P_2$ iff P_1 implements P_2 (i.e, is less nondeterministic than P_2). The set **P** could for instance be the set of all CSP processes and the relation $\leq$ the one defined by the inclusion relation on the failure set semantics of these processes. Let us assume some operations are defined on the set of processes **P**, for instance parallel composition, renaming and hiding. The exact set of operations is not important in this general context, what matters is that these operations are monotonic with respect to the relation $\leq$. For instance if parallel composition ($\|$) is among the set of operations considered, one requires that if

$$P_1 \leq P_2$$

then, for any $P \in \mathbf{P}$

$$P \parallel P_1 \leq P \parallel P_2.$$

Consider now a network composed of a large number (n) of identical processes $P \in \mathbf{P}$ and one special process P_0. We assume these processes are linked by parallel composition or, more generally, by a combination of parallel composition and other operations such as renaming and hiding. One thus has

$$N_n = P_0 \odot \underbrace{P \odot P \odot \ldots \odot P}_{n}$$

where $\odot$ represents the combination of operations linking the processes (often a combination of parallel composition, renaming and hiding). The problem is to prove that, for any n, N_n satisfies some specification S which is also given as a process. In other words, one has to prove that for all n,

$$N_n \leq S.$$

The idea we propose is that this can be done using a process I which we call a *network invariant.* To be a network invariant, I should satisfy the following conditions:

$$P_0 \leq I \tag{1}$$

$$I \odot P \leq I \tag{2}$$

If the relations 1 and 2 are satisfied, it follows by induction and the fact that process composition is monotonic with respect to $\leq$ that, for all n, $N_n \leq I$. Thus, if I is chosen such that $I \leq S$, this gives us a way of proving that for all n, $N_n \leq S$. If the relation $\leq$ is checkable algorithmically (which is the case for most definitions of finite-state processes), this gives us an algorithmic verification method once the network invariant I is known.

Before showing how this method can be used with a particular type of process, let us see how it can also be used for the verification of circular networks of processes. A circular network of processes is one in which all processes are similar and are linked in a ring. Our method assumes the processes to be identical and composed in an identical way. Thus, it can easily be used to reason about a chain of processes, but there is no way of closing the chain. The trick is to use a (slightly) different process to close the chain. Note that this is even often required by the problem specification given that the ring cannot always be perfectly symmetrical. For instance in a token passing ring, some process initially has the token. Thus a circular network C_n has the form

$$C_n = P_\ell \odot_\ell \underbrace{P \odot P \odot \ldots \odot P}_{n}.$$

To prove that such a process satisfies a specification S, we proceed as follows. We first use a network invariant I and show that

$$\underbrace{P \odot P \odot \ldots \odot P}_{n} \leq I$$

by proving that

$$P \leq I \tag{3}$$

$$I \odot P \leq I \tag{4}$$

We then simply check that

$$P_\ell \odot_\ell I \leq S$$

which proves the desired property.

3 A Process Theory

In this section, we present the process theory on which our examples are based. It is a variant of TCSP [Hoa85]. It is however not the only possible choice. Any process semantics for which process composition operations are monotonic is adequate.

We use a limited process description language. We consider finite automata as basic processes and parallel composition, renaming and hiding as the only operations on processes. This is not really a restriction since finite state CCS or TCSP programs can be systematically transformed into transition systems [Mil80], [Old85].

Processes are defined over an alphabet of communication actions Σ. In addition to transitions corresponding to actions, processes can take silent transitions that will be labeled by the silent (internal) action τ. A basic process P is a a finite-state transition system, precisely a quadruple

$$P = (\Sigma, S, \rho, s_0)$$

where

- Σ is the alphabet of actions.
- S is a finite set of states,
- $\rho : S \times (\Sigma \cup \{\tau\}) \to 2^S$ is a transition relation that for each state and action gives the possible next states,
- $s_0 \in S$ is the initial state of the process.

We consider three operations on processes: parallel composition, renaming and hiding. Parallel composition (denoted $\|$) corresponds to the concurrent execution of two processes with handshaking on events that are common to both processes. Renaming allows us to modify the alphabet of a process and hence to choose the events on which handshaking will take place in concurrent composition. We denote the process P in which a is renamed to b by $P[b/a]$. Hiding transforms external actions into the internal action. The process P in which all actions in the set A are hidden is denoted by $P\backslash A$.

It is straightforward to give operational semantics to our process theory in terms of automata. We are however interested in a more abstract semantics. To choose our semantic domain, we first consider what properties of processes we want to observe on the global system. Three items seem natural:

- The sequences of actions that can be performed;
- The possibility of deadlock;

- The possibility of divergence (infinite sequence of internal actions).

So, our semantics for processes should give us information about these items. Moreover, we want our semantics to be fully abstract with respect to the operations we have defined. That is, we want to be able to determine the semantics of a composed process from its parts and we want our semantics to be the weakest one compatible with this requirement and the elements we want to observe.

In [Mai86], [Hen87], it is shown that if we are not very discriminating about divergence, the process semantics that satisfy our requirement are essentially failure semantics [Hoa85]. The failures of a process are the pairs (s, X) where s is a sequence of external actions of the process and X is a set of actions the process can refuse after executing s. We do not want a treatment of divergence as radical as the one in [BHR84] [Mai86], so we add explicit divergence points to our semantics [Old85]. A divergence point is a pair $(s, \uparrow)$ where s is a sequence of external actions and $\uparrow$ is a special symbol indicating that the process can diverge after executing the sequence s. So, for a process defined on the alphabet Σ the semantic domain will be

$$2^{FAILURES_\Sigma} \times 2^{DIVERGENCES_\Sigma}.$$

In other words, the semantics of a process is a set of failures and a set of divergence points. The semantics of a given process can be determined in a standard way [BHR84], [Hoa85], [Old85].

Finally, we need an order on our semantic domain that represents the implementation relation mentioned in our general framework. Let P_1 and P_2 be two processes whose semantics are respectively (F_1, D_1) and (F_2, D_2). Then we will have that

$$P_1 \leq P_2$$

(P_1 implements P_2) iff

$$F_1 \subseteq F_2 \text{ and } D_1 \subseteq D_2{}^1.$$

It is easy to check that the three operations we use on processes (parallel composition, renaming and hiding) are monotonic with respect to the relation $\leq$ on our semantic domain.

Note that checking if two finite-state processes are related by the relation $\leq$ can be done algorithmically [KS83]. It is however a PSPACE complete problem. This complexity is due to the particular semantics we have used which was chosen to be theoretically sound. In practice, one could often use a stronger semantics (bisimulation or observational equivalence [Mil80], [vG86]) which can be checked more efficiently [KS83].

As a last remark, let us note that it is also possible to add to a process a restriction on its infinite behaviors (for instance a fairness condition stating that some events must

[1]Most orders that have been defined on semantic domains similar to ours are in the opposite direction which is more natural when dealing with semantic issues. We choose the direction that corresponds to implementation (less nondeterminism).

appear infinitely often). This could be done as described in [Par85] or [ALW89] by adding to the process a language of infinite words (described by an ω-regular expression or by a Büchi automaton) specifying this requirement. In this case, the semantic domain should be extended to incorporate the infinite sequences that can be produced by the processes.

4 Finding Network Invariants

A natural question to ask is whether invariants always exist. Unfortunately the answer is negative. Let us call the *verification problem* the problem of determining in the framework outlined above if for finite-state processes P_0, P and S we have that for all n,

$$N_n = P_0 \odot \underbrace{P \odot P \odot \ldots \odot P}_{n} \leq S$$

It is rather straigtforward to adapt the proof given in [AK86] to show the following theorem.

Theorem 4.1 *The verification problem is complete in co-RE.*

One then has:

Corollary 4.2 *There are positive instances of the verification problem for which finite-state invariants do not exist.*

Indeed, if positive instances of the verification problem always had finite-state invariants, the verification problem would be in RE which contradicts Theorem 4.1. A weaker related question is to determine if there exists an invariant for a verification. It might be interesting to know that there is no invariant even though the property might be true. This question is also undecidable.

Theorem 4.3 *Determining if an instance of the verification problem admits an invariant is complete in RE.*

All the preceding results are bad news. However, they are general results and do not imply that invariants cannot be found in a lot of interesting cases. In the next section we will give invariants for a token passing mutual exclusion protocol and for a buffer.

The buffer example will however require some special ingenuity since it is known [SCFG82] that buffers cannot be characterized in a finite-state framework. The solution is provided in [Wol86] where it is shown how the problem can be solved if one can assume that the processes are *data-independent.* A process is data-independent if its behavior does not rely on the value of the data objects it manipulates. More precisely, a process is data-independent with respect to the data read at one of its input ports if, when the value of this data is modified, the possible behaviors of the process are unchanged except for a corresponding change in the values of the data at its input and output ports.

Given this property, one can specify a buffer in a finite-state framework as follow. The property one requires of a buffer is that if it is fed an infinite sequence of distinct messages, then it will output the same sequence of messages. Under the hypothesis of data-independence, one can consider a variant of the buffer that only distinguishes between three different message values m_0, m_1 and m_2. It is then sufficient to show that if the buffer is fed a sequence of the form

$$m_0^\star; m_1; m_0^\star; m_2; m_0^\omega,$$

where $\star$ represents finite repetition and ω represents infinite repetition, it output sequence is of the same form. We will use this specification of a buffer in the next section.

5 Examples

In this section, we present two examples of the use of our method. The first is an arbitrary size buffer composed of elementary buffers. The second is a token passing mutual exclusion protocol. In the full paper, we will also consider an arbitrary size stack built from elementary stack cells.

5.1 A Buffer

Let P be the elementary buffer represented in Figure 1. To enable us to use the theory of data-independence described in Section 4, we have chosen a version of the elementary buffer that can distinguish between three different types of messages: m_0, m_1 and m_2.

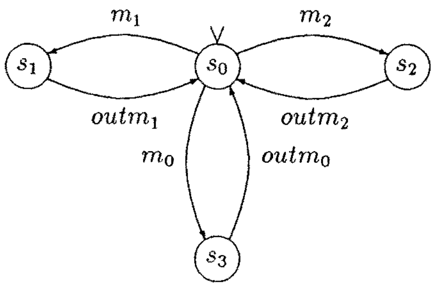

Figure 1: The elementary buffer P

Now, an n place buffer P_n can be obtained by connecting together n instances of P. This is done by running the elementary buffers in parallel and connecting the outputs of each buffer to the inputs of the next one. Precisely, we have that

$$P_n = \underbrace{P \odot P \odot \ldots \odot P}_{n},$$

where ($i = 0, 1, 2$)

$$P \odot P = (P[lk_i/outm_i] \parallel_{\{lk_i\}} P[lk_i/m_i]) \backslash \{lk_i\}$$

Note that only the first and last processes P communicate with the external environment.

To prove that P_n behaves like an n places buffer, me must show that if its input is the process P_0 of Figure 2, then its output behavior is also of the form described by P_0. Formally, we should prove that

$$P_0 \odot P_n \leq P_0. \tag{5}$$

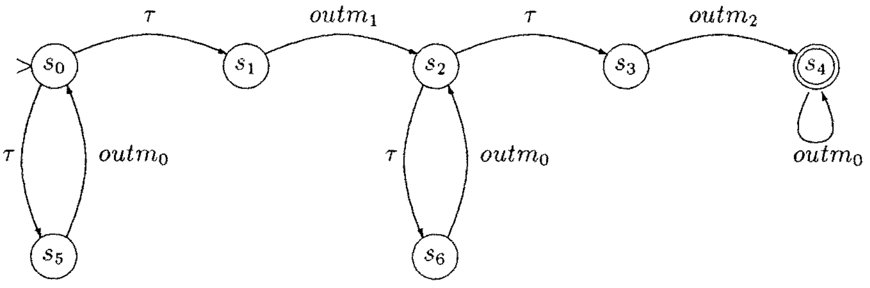

Figure 2: The specification P_0 of the buffer behavior

To prove (5), we use the network invariant $I = P_0$. We must thus check that

$$P_0 \leq I$$

and that

$$I \odot P \leq I.$$

The first condition is immediate and the second one can easily be checked by computing the process $P_0 \odot P$. Note that to check the liveness property of the buffer (if it is fed an infinite sequence of messages, then it outputs an infinite set of messages) it is sufficient to consider the process P_0 where only the behaviors going infinitely often through state s_4 are admissible. One then checks that this property is preserved by composition with the process P.

5.2 A Token Passing Mutual Exclusion Protocol

Consider a ring obtained by linking n copies of the process described in Figure 3 and one copy of the same process with initial state s_1. It is easy to see that such a combination of processes implements a token passing mutual exclusion algorithm. The process with initial state s_1 has the token initially and it can either go through its critical section once or pass the token to the next process. The other processes behave in a similar way, except that initially they don't have the token. The unicity of the token ensures mutual exclusion.

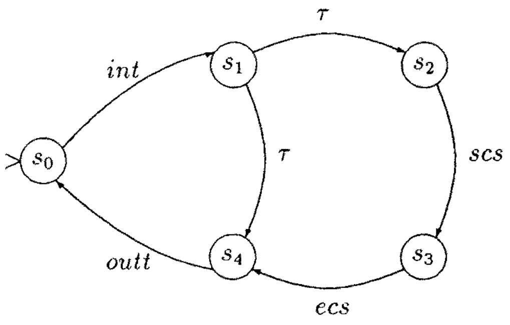

Figure 3: A participant P in the token passing mutual exclusion algorithm

Formally, the network will be composed of n identical processes P and one process P_ℓ identical to P except for its initial state which is s_1. We thus have

$$C_n = P_\ell \odot_\ell \underbrace{[P \odot P \odot \ldots \odot P]}_{n}$$

where

$$P \odot P = [P[lk/outt] \parallel_{\{lk\}} P[lk/int]] \backslash \{lk\}$$

and

$$P_\ell \odot_\ell P = [P_\ell[int/outt, outt/int] \parallel_{\{int, outt\}} P] \backslash \{int, outt\}.$$

Mutual exclusion is guaranteed if the operations scs and ecs strictly alternate. Thus, the specification of the correct behaviors of C_n can be represented by the graph of Figure 4.

To verify that C_n is a realization of S, we must find a network invariant I such that the three following conditions are satisfied.

$$P \leq I \tag{6}$$

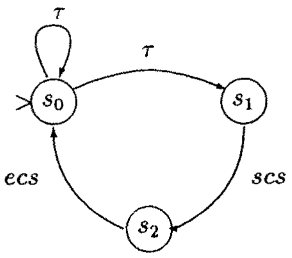

Figure 4: The specification S

$$I \odot P \leq I \tag{7}$$

$$P_\ell \odot_\ell I \leq S \tag{8}$$

Such an invariant is given in Figure 5. The invariant essentially represents the joint behavior of any number of elementary processes P. The only difficulty in finding the invariant was that when the network is not closed by the process P_ℓ, nothing guarantees the unicity of the token and hence the mutual exclusion. This problem was solved by chosing the invariant such that all behaviors are possible (essentially nothing is specified) if more than one token is in the network. Given that when the process P_ℓ is composed with the invariant only one token is introduced, this is sufficient to prove the mutual exclusion property. It is indeed easy to check that the invariant I satisfies the required properties given above.

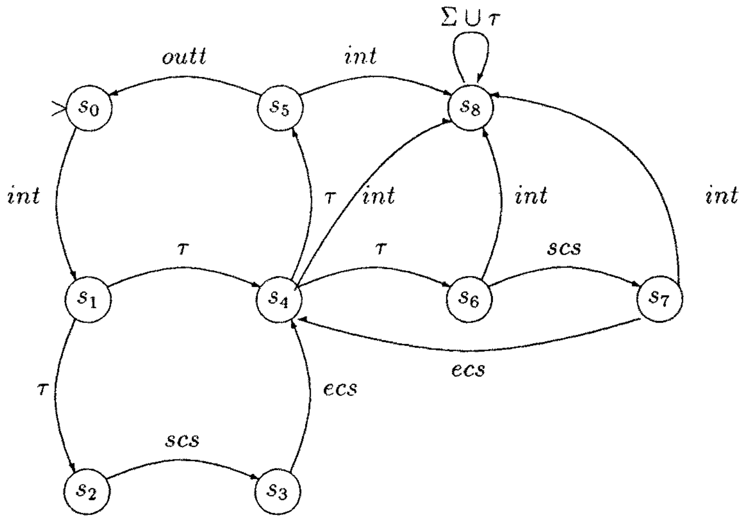

Figure 5: The invariant I

6 Conclusion and Comparison with Other Work

We have presented an alternative method for reducing the verification of a network composed of an unbounded set of identical processes to a finite-state problem. The main evidence we can give for the usefulness of our method are the examples we handle. For instance, we know of no other similar verification of a buffer composed of a arbitrary number of elementary buffers. The fact that we were able to handle the token ring example shows the flexibility of the method.

Another advantage of our method is that it can piggy back on a verification system for finite-state processes. No specific implementation is necessary. We have presented the method in a particular framework suitable for our examples. Other framework are certainly possible. One could also imagine using the same method with processes described by logical formulas, for instance temporal logic formulas [Pnu81], [EH86], [Wol83], [GS86].

Having to find an invariant can be seen as a drawback of our method. It is also an advantage. The invariant captures the reason for which the network behaves as expected. In this sense it helps to understand the algorithm being used. We believe it is unlikely that fully automatic methods can solve interesting cases of reasoning about networks built from large numbers of identical processes. The undecidability results we give in Section 4 definitely support this point of view.

Other work on the verification of networks of large numbers of processes includes [CGB86], [CG87], [GS87] and [KM89]. Also, in [AE89] a method for synthesizing networks of identical processes from temporal logic is presented. The results in [KM89] were developed independently of (and simultaneously with) ours and turn out to be quite similar. This paper also presents an induction theorem as a method for proving properties of large sets of processes. However the form of the theorem, the process theory with which it is used and the examples are different from the ones considered here.

In [CGB86], the method presented has its origin in model checking [CES86] and is based on a result that proves that for a given logic, if a formula is true of a structure, then it is also true of all structures that can be put in some correspondence with that structure. The crucial point is to prove that there is such a correspondence between the structure corresponding to n processes and the one corresponding to a small fixed number k of processes. This step is however manual.

The approach presented in [CG87] is closer to ours but has several weaknesses. First, it only deals with concurrent composition of CCS processes. The fact that renaming and hiding are not considered limits the type of examples that can be solved. Second, the equivalence relation between processes that is used is not a congruence with respect to process composition. This substantially complicates the method. Third, it requires the discovery of a "process closure". Actually this process closure is very much like our network invariants and needs the help of the user to be discovered. We believe that the verification method as we have presented it simplifies and extends the applicability of the one described in [CG87]

In [GS87], the method presented is fully algorithmic. It has several drawbacks. First, the complexity of the algorithms being used is very high (double exponential time). Second, the networks of processes that can be checked are the parallel composition of identical CCS processes. Renaming and hiding as we use them are unavailable and thus it is for instance impossible to restrict a process to only communicate with its neighbors. An extension of the method to handle this type of communication is discussed, but it is no longer a decision procedure and often will not work.

Acknowledgement

The theorems of Section 4 arose from discussions with Costas Courcoubetis.

References

[AE89] P. C. Attie and E.A. Emerson. Synthesis of concurrent systems with many similar sequential processes. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, pages 191–201, Austin, January 1989.

[AK86] K. Apt and D. Kozen. Limits for automatic verification of finite-state concurrent sytems. *Information Processing Letters*, 22(6):307–309, 1986.

[ALW89] M. Abadi, L. Lamport, and P. Wolper. Realizable and unrealizable concurrent program specifications. to appear, 1989.

[BHR84] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(7):560–599, 1984.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[CG87] E. M. Clarke and O. Grümberg. Avoiding the state explosion problem in temporal logic model-checking algorithms. In *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pages 294–303, Vancouver, British Columbia, August 1987.

[CGB86] E. M. Clarke, O. Grümberg, and M. C. Browne. Reasoning about networks with many identical finite-state processes. In *Proc. 5th ACM Symposium on Principles of Distributed Computing*, pages 240–248, Calgary, Alberta, August 1986.

[EH86] E.A. Emerson and J.Y. Halpern. Sometimes and not never revisited: On branching versus linear time. *Journal of the ACM*, 33(1):151–178, 1986.

[GS86] S. Graf and J. Sifakis. A logic for the description of non-deterministic programs and their properties. *Information and Control*, 68(1-3):254–270, 1986.

[GS87] S. German and A.P. Sistla. Reasoning with many processes. In *Proc. Symp. on Logic in Computer Science*, pages 138–152, Ithaca, June 1987.

[Hen87] M. Hennessy. Why testing equivalence is natural? Handwritten Note, April 1987.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[KM89] R. P. Kurshan and M. McMillan. A structural induction theorem for processes. In *Proceedings of the Eigth ACM Symposium on Principles of Distributed Computing*, Edmonton, Alberta, August 1989.

[KS83] P. C. Kanellakis and S. A. Smolka. CCS expressions, finite state processes and three problems of equivalence. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 228–240, Montreal, Quebec, August 1983.

[Mai86] Michael G. Main. Demons, catastrophies and communicating processes. Technical Report CU-CS-343-86, Department of Computer Science, University of Colorado, July 1986.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science.* Springer Verlag, Berlin, 1980.

[Old85] E.R. Olderog. Process theory: Semantics, specification and verification. In *Proc. Advanced School on Current Trends in Concurrency*, pages 442–509, Berlin, 1985. Volume 224, LNCS, Springer-Verlag.

[Par85] J. Parrow. *Fairness Properties in Process Algebra.* PhD thesis, University of Uppsala, Sweden, 1985.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[SCFG82] A. Sistla, E.M. Clarke, N. Francez, and Y. Gurevich. Can message buffers be characterised in linear temporal logic. In *Proceedings of the First Symposium on Principles of Distributed Computing*, Ottawa, Canada, August 1982.

[vG86] R. J. van Glabbeek. Notes on the methodology of CCS and CSP. Technical Report CS-R8624, CWI Amsterdam, 1986.

[Wol83] P. Wolper. Temporal logic can be more expressive. *Information and Control*, 56(1-2):72–99, 1983.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming*, pages 184–192, St. Petersburgh, January 1986.