May 1994

State Minimization for Concurrent System Analysis Based on State Space Exploration

Inhye Kang
University of Pennsylvania

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-94-22.

State Minimization for Concurrent System Analysis Based on State Space Exploration

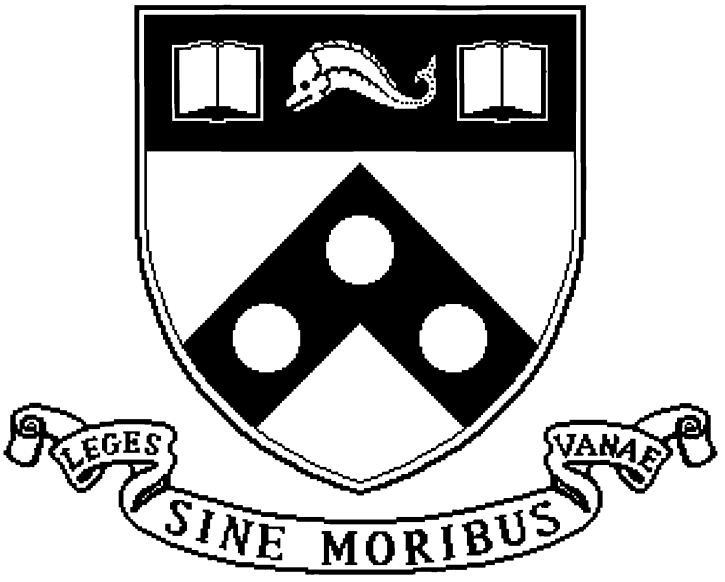
Abstract
A fundamental issue in the automated analysis of concurrent systems is the efficient generation of the reachable state space. Since it is not possible to explore all the reachable states of a system if the number of states is very large or infinite, we need to develop techniques for minimizing the state space. This paper presents our approach to cluster subsets of states into equivalent classes. We assume that concurrent systems are specified as communicating state machines with arbitrary data space. We describe a procedure for constructing a minimal reachability state graph from communicating state machines. As an illustration of our approach, we analyze a producer-consumer program written in Ada.

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-94-22.

State Minimization for Concurrent System Analysis Based on State Exploration

MS-CIS-94-22
LOGIC & COMPUTATION 80

Inhye Kang
Insup Lee



University of Pennsylvania
School of Engineering and Applied Science
Computer and Information Science Department
Philadelphia, PA 19104-6389

May 1994

State Minimization for Concurrent System Analysis Based on State Space Exploration *

Inhye Kang and Insup Lee
Department of Computer and Information Science
University of Pennsylvania
Philadelphia, PA 19104-6389

to appear in Proc. of COMPASS '94

Abstract

A fundamental issue in the automated analysis of concurrent systems is the efficient generation of the reachable state space. Since it is not possible to explore all the reachable states of a system if the number of states is very large or infinite, we need to develop techniques for minimizing the state space. This paper presents our approach to cluster subsets of states into equivalent classes. We assume that concurrent systems are specified as communicating state machines with arbitrary data space. We describe a procedure for constructing a minimal reachability state graph from communicating state machines. As an illustration of our approach, we analyze a producer-consumer program written in Ada.

1 Introduction

One of the most prohibitive barriers in automatic analysis based on state space exploration of a concurrent system is *state explosion* [4, 13]. Two major sources of state explosion are process composition and data space size. The state space of a system grows exponentially with the number of subsystems because its size is proportional to the product of the number of states within each subsystem. In addition, since a state is defined by the values of the variables used in a system, the number of states depends proportionally on the size of data space.

For dealing with state explosion due to process composition, compositional analyses of finite state systems have been developed [13, 5]. To deal with the large data space

*This research was supported in part by DARPA/NSF CCR90-14621 and NSF CCR93-11622.

problem, Jonsson and Parrow developed a technique to cluster states into equivalent classes [7]. Their approach, however, seems to have a limited use in analyzing realistic concurrent systems since they assume that control is data value independent. In most systems, data values are used to determine control flow.

In this paper, we address the explosion problem caused by large data space size. We propose a different approach from [7] to handle *data-dependent* systems. Our approach is to cluster states that are bisimilar but have different data values into an equivalent class. For example, suppose that a system includes an integer value in its state. If we assume that an integer is stored in four bytes, then the variable can have one of 2^{32} possible values. Thus, this variable can increase the size of the state space by multiplicative of 2^{32}. However, not all of the states with different values may need to be distinguished. For instance, if the program contains the statement "if $x > 3$ then send a message to channel a", then the values of x can be divided into two classes, $\{v|v > 3\}$ and $\{v|v \leq 3\}$. Here, it may be good enough to treat it as if there are only two possible values for the variable x.

For the specification of concurrent systems, we have extended Communicating State Machines (CSMs) by Shaw [12] with composition and one-to-many communication. Each CSM has local variables whose values are from *arbitrary* data domains, and is a transition system in which transitions are guarded by enabling conditions over variables. Communication is one-to-many synchronous communication such that the value sent by the sending CSM is received by all the receiving CSMs. We model an execution of CSMs as a labeled transition system with possibly an infinite number of states. We also model the reachable state space of CSMs as a labeled transition system, which is the union of all labeled transition systems that represent the executions of the CSMs. Our goal is to develop a technique to minimize the size of a labeled transition system, which represents the reachable state space of CSMs.

Our approach is inspired by the minimization algorithm developed by Bouajjani *et al.* [2]. Their algorithm efficiently constructs the minimal reachability graph of an unlabeled transition system. However, the unlabeled transition system is not expressive enough to describe communicating concurrent systems. Our minimization algorithm extends their algorithm to a labeled transition system generated from CSMs. In particular, our algorithm generalizes their algorithm with multiples initial states and multiple relations (i.e., labels). Similar to their algorithm, our minimization procedure does not always terminate. However, we believe it to be powerful enough to handle many interesting communicating systems with an infinite number of reachable states. As a continuing work, we have identified a set of fairly general sufficient restrictions on the syntax of CSMs which guarantee termination.

The rest of the paper is organized as follows. In Section 2, we overview other methods

related to our work. Section 3 defines CSMs and explains the generation of a labeled transition system from CSMs. Section 4 describes how to minimize a labeled transition system and presents an example using CSMs. In Section 5, we apply our approach to a simple Ada program to illustrate its potential in concurrent program analysis. Section 6 concludes with discussion of the future work.

2 Related Work

There have been several work that address the problem of state explosion in the analysis of concurrent systems. One approach to controlling the state explosion in process composition is compositional analysis [5, 13]. In [5], the analysis of $P||Q$ is reduced to the analysis of each component process, say P, with a simpler process Q'. The process Q' is called an environment and is simpler than Q by hiding details not relevant to interaction with P. In this approach, the complexity of the analysis of a property depends not on the size of the composite process but on the size of each component process multiplied by the size of its environment process. Yeh and Young [13] describes an interesting application of process algebra in the compositional analysis of concurrent systems. Their approach is to construct a smaller reachability graph of a composite process using a divide-and-conquer strategy. The reachability graph of a subsystem is replaced by an equivalent but smaller graph, and the smaller graphs of the subsystems are combined to form a larger system. These two approaches do not directly address the explosion problem due to data space.

Jonsson and Parrow [7] provide a technique to change a program with infinite states due to infinite variable space size into an equivalent finite state program. The main idea is to represent the data values of a variable using a finite number of symbols. This technique, however, is limited to programs in which control flow is data-independent.

There are several general state minimization algorithms for labeled transition systems that have been developed [8, 11, 6]. These algorithms require the generation of the entire state space, including unreachable states. Thus, they can be applied only to systems with a finite, relatively small state space. It would be desirable to explore only the reachable portion of the state space. Bouajjani *et al.* have developed such an algorithm to find the minimal reachability graph for unlabeled transition systems [2]. Their algorithm performs reachability analysis and minimization simultaneously. This algorithm is very effective when the reachable portion is much smaller than the full state space. Furthermore, the algorithm deals with an infinite data space by representing states symbolically. Alur *et al.* [1] have applied the algorithm to timed transition systems without data variables.

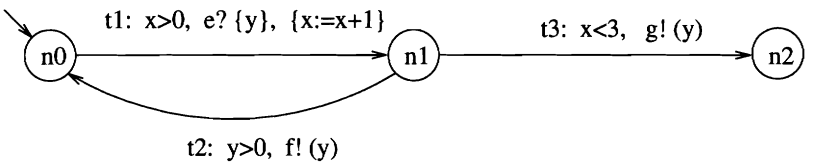


Figure 1: A simple CSM

3 CSMs: Communicating State Machines

As Shaw [12] points out, state machines are easier to understand, visualize and implement. We thus adopt and extend CSMs with composition to provide a general framework of specification and analysis of concurrent systems. We use CSMs because of its generality and simplicity in describing concurrent systems. As an illustration, we translate a procedure-consumer program written in Ada to CSMs in Section 5.

CSMs are state machines which communicate synchronously over channels. CSMs have no shared variables, and execute independently except when they communicate through channels. Communication over a channel is synchronous so that all CSMs connected by the same channel are required to engage in communication simultaneously. For each channel e, there is a domain of possible values, called $dom(e)$, that can be communicated through e. A send action $e!(exp)$ denotes the sending of a message exp over the channel e, whereas a receive action $e?X$ denotes the reception of a value through the channel e. CSMs are compositional, i.e., the composition of several CSMs results in a CSM. Transitions within a CSM are labeled with enabling conditions and communication actions. Figure 1 shows a simple CSM.

Definition 3.1 *A CSM is a tuple* $M = \langle \Sigma, N, n_0, V, T \rangle$, *where* Σ *is a finite set of connected channels,* N *is a finite set of nodes,* $n_0 \in N$ *is the initial node,* V *is a finite ordered set of variables, and* T *is a finite set of transitions.*

In a CSM, each variable x has the value domain $dom(x)$ and the set of possible initial values $I(x)$ such that $I(x) \subseteq dom(x)$. The domain of a variable and the set of possible initial values can be infinite. In the example shown in Figure 1, the variables x, y are for integers and we assume that their initial values are the interval $(0, 10)$. (The initial values are used later in Section 4.) A transition in T is of the form (n_1, c, a, h, n_2), where n_1 is the source node, c is an enabling condition over variables, a is an action, h is the set of assignments over variables, and n_2 is the target node.

Unlike CSMs proposed by Shaw [12], our CSMs are compositional [9]. When two CSMs are composed into a CSM, transitions of the CSMs which are not using a shared channel are interleaved. On the other hand, transitions using a shared channel are synchronized and composed as follows:

- Two receive transitions $(n_1, c_1, e?X_1, h1, n_1')$ and $(n_2, c_2, e?X_2,\ h2, n_2')$ result in a receive transition $((n_1, n_2), c_1 \wedge c_2, e?X_1 \cup X_2, h1 \cup h2, (n_1', n_2'))$.

- A send transition $(n_1, c_1, e!(exp), h1, n_1')$ and a receive transition $(n_2, c_2, e?\{x_1, .., x_l\}$, $h2, n_2')$ result in a send transition $((n_1, n_2), c_1 \wedge c_2, e!(exp), h1 \cup h2 \cup \{x_i := exp | 1 \leq i \leq l\}, (n_1', n_2'))$.

In modeling the execution of a CSM, a state is represented by a pair (n, v), where n is a node and v is the valuation of data variables. The execution of a CSM starts at a state (n_0, v_0), where n_0 is the initial node and v_0 is a valuation such that for each variable x, $v_0(x)$ is in $I(x)$. A transition (n_1, c, a, h, n_2) can be taken from the current state (n_1, v_1) only if the current valuation v_1 satisfies the condition c. The effect of taking the transition (n_1, c, a, h, n_2) from a state (n_1, v_1) is a state (n_2, v_2), where v_2 is any valuation in a set of possible valuations, called $f(v_1, t)$, defined as follows:

$$\begin{aligned} f(v_1, t) = \{v_2 |\ & \forall x \in ivar(a).v_2(x) \in dom(event(a)) \\ & \wedge\ \forall x, y \in ivar(a).v_2(x) = v_2(y) \\ & \wedge\ \forall x \in V - ivar(a).v_2(x) = h(v)(x)\} \end{aligned}$$

where $event(a)$ is the channel associated with the action a and $ivar(a)$ is a set of variables whose values might change by the communication action a; that is, $ivar(a)$ is a set of variables named in the action a if a is a receive action; $ivar(a)$ is an empty set if a is a send action.

Definition 3.2 *An execution of a CSM $M = \langle \Sigma, N, n_0, V, T \rangle$ is a finite or infinite sequence of the form*

$$s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots$$

where $s_i = (n_i, v_i)$ satisfying

1. *Initiality: $v_0(x) \in I(x)$ for every $x \in V$.*

2. *Succession Constraint: for each i, there exists transition t_i in T from n_i to n_{i+1} such that v_i satisfies the enabling condition of t_i and $v_{i+1}(x) \in f(v_i, t_i)(x)$.*

When we analyze a concurrent system, we are usually interested in observable behaviors, not values of internal variables. A behavior is a sequence of observable events such as communication, input and output. In CSMs, we define a behavior to be a sequence of communication events ignoring their message values. Thus, a behavior of a CSM can be obtained from an execution as follows: for an execution $s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \cdots$, the corresponding behavior is a sequence of events $e_0, e_1, e_2, ...$ such that e_i is the channel used by transition t_i.

Definition 3.3 *A* labeled transition system *is defined as a tuple* $\mathcal{S} = \langle \Sigma, S, S_0, \rightarrow \rangle$, *where* Σ *is a set of events,* S *is a set of states,* S_0 *is a set of the initial states, and* $\rightarrow \subseteq S \times \Sigma \times S$ *is a transition relation.*

Note that it is easy to define a labeled transition system from an execution of a CSM. Furthermore, it is possible to define the labeled transition system corresponding to all the executions of the CSM.

4 Minimization of the State Space of CSMs

We show how to minimize the state space of a given CSM. Here, we construct a minimal *transition system* from a given CSM directly without generating the entire state space of the CSM. Not having to generate the entire state space first is important especially for a state space with a large or infinite number of states.

Bouajjani *et al.* have developed an efficient algorithm that minimizes an unlabeled transition system without explicitly generating all the states. This algorithm is called the BFH algorithm in the rest of this paper. Figure 2 shows the BFH algorithm modified to allow multiple initial states.

The basic idea of minimizing a transition system is to find a partition of states such that all the states in each class of the partition are bisimilar and all bisimilar states are in the same class. Starting from the class consisting of the entire states as the sole member of the initial partition, the BFH algorithm tries to iteratively split classes in the current partition until it is no longer possible. The splitting procedure keeps states in the same class until they are shown to be non-bisimilar. Such a class is called *stable* with respect to the current partition in the algorithm. In other words, for a given initial partition ρ_0, the algorithm repeatedly split classes that are not stable with respect to the current partition until the coarsest stable partition is found. The result of the algorithm is the coarsest stable partition which is equal to the greatest bisimulation refining ρ_0.

The algorithm shown in Figure 2 uses the following symbolic operators for a partition ρ and a class X in ρ:

- $post_\rho(X)$ denotes the set of classes of ρ which contains at least one state immediately reachable from a state of X.
- $pre_\rho(X)$ denotes the set of classes of ρ which contains at least one state from which a state of X is immediately reachable.
- $split(X, \rho)$ divides X into the largest subclasses which are all stable with respect to ρ.

algorithm Minimization
  $\rho := \rho_0$; % ρ_0: the initial partition, ρ : the current partition
  $R := \{X \in \rho | Q_0 \cap X \neq \emptyset\}$; % R : the set of explored reachable classes
  $R_s := \emptyset$; % R_s : the set of explored stable classes
  while $R \neq R_s$ **do**
    choose X in $R - R_s$;
    $N := split(X, \rho)$;
    if $N = \{X\}$ **then**
      $R_s := R_s \cup \{X\}$;
      $R := R \cup post_\rho(X)$;
    else
      $R := R - \{X\}$;
      $R := R \cup \{Y \in N | Y \cap Q_0 \neq \emptyset\}$; % Extension with many initial states
      $R_s := R_s - pre_\rho(X)$;
    $\rho = (\rho - \{X\}) \cup N$;
  end while
end algorithm

Figure 2: Minimization Algorithm

We note the algorithm may not always terminate. It terminates only when the greatest bisimulation has finite number of equivalence classes. The algorithm is said to be efficient in the sense that only reachable portions of the state space are explored.

The notions of stability and bisimulation have been defined for an unlabeled transition system in [2]. In order to apply the BFH algorithm to labeled transition systems, we need to define stability and bisimulation for a labeled transition system. We define the notion of stability for a labeled transition system as follows:

Definition 4.1 *A class X is said to be stable with respect to a partition ρ iff for every class $Y \in \rho$, for every event $e \in \Sigma$, if for some state s_1 in X, there exists $s_1' \in Y$ such that $s_1 \xrightarrow{e} s_1'$, then for every other state s_2 in X, there exists $s_2' \in Y$ such that $s_2 \xrightarrow{e} s_2'$.*

As for the notion of bisimulation, we use the well-known definition of bisimulation [10]:

Definition 4.2 *Given a labeled transition system $S = \langle \Sigma, S, S_0, \rightarrow \rangle$, a binary relation $\rho \subseteq S \times S$ is a bisimulation iff*

$$\begin{aligned} &\forall (s_1, s_2) \in \rho.\ \forall e \in \Sigma. \\ &\quad \forall s_1'.(s_1 \xrightarrow{e} s_1' \Rightarrow \exists s_2'.(s_2 \xrightarrow{e} s_2' \wedge (s_1', s_2') \in \rho)) \wedge \\ &\quad \forall s_2'.(s_2 \xrightarrow{e} s_2' \Rightarrow \exists s_1'.(s_1 \xrightarrow{e} s_1' \wedge (s_1', s_2') \in \rho)). \end{aligned}$$

It is also true that for a given initial partition ρ_0, the coarsest stable partition is equal to the greatest bisimulation refining ρ_0 with these notions of stability and bisimulation [9]. Since the BFH algorithm applied to a labeled transition system gives the coarsest stable partition, we can compute the greatest bisimulation of a labeled transition system using the algorithm.

4.1 CSM State Minimization

To generate a minimal transition system from a given CSM using the BFH algorithm, we define the initial partition and three operations on CSMs.

The Initial Partition. Suppose a CSM $M = \langle \Sigma, V, N, n_0, T \rangle$, where $V = \{x_1, \ldots, x_k\}$. Let D be the data space, i.e., $D = dom(x_1) \times \cdots \times dom(x_k)$. We define the initial partition of the whole state space to be $\rho_0 = \{\{(n, v) | v \in D\} | n \in N\}$, not $\{N \times D\}$. Although it is possible to define the initial partition to be $\{N \times D\}$, it becomes too complex to define necessary operators to apply the BFH algorithm. Furthermore, it is natural to equate states with the same node since an enabling condition is a criterion for dividing the data space. Let Z represent a subset of D. We represent an equivalence class $\{(n, v) | v \in Z\}$ as (n, Z). For example, ρ_0 is represented as $\{(n, D) | n \in N\}$.

Three Operators. Let ρ be a partition of $N \times D$. Let X and Y be (n_1, Z_1) and (n_2, Z_2), respectively. First, we define the operator $post_\rho(X)$ equal to the set S of classes in ρ such that for each class $W \in S$, there is a state in X that can go to a state in W. Suppose there is a transition $t = (n_1, c, a, h, n_2)$. From X, only the states in X that satisfy the enabling condition c of t (i.e., $(n_1, Z_1 \cap c)$) can proceed to n_2 with the transition t. As a result, the valuation is changed by action a and statement h as explained in Section 3.

The set of possible valuations of Z_1 as the result of taking the transition t is defined as follows:

If $Z_1 \cap c = \emptyset$ then $f(Z_1, t) = \emptyset$;
Otherwise,

$$f(Z_1, t) = \{v | \ \forall x \in ivar(a).\ v(x) \in dom(event(a)) \wedge \\ \forall x, y \in ivar(a).\ v(x) = v(y) \wedge \\ \forall x \in (V - ivar(a)).\ v(x) \in h(Z_1 \cap c)(x)\ \}$$

Here, we say that every state in a class $(n_2, f(Z_1, t))$ is immediately reachable from (n_1, Z_1) via t. In other words, if $Z_2 \cap f(Z_1, t)$ is not empty then Y is included in $post_\rho(X)$. Therefore, the operator $post_\rho$ is defined as follows:

$$post_\rho((n_1, Z_1)) = \{(n_2, Z_2) \in \rho | \exists t \text{ from } n_1 \text{ to } n_2 \in T. f(Z_1, t) \cap Z_2 \neq \emptyset\}$$

Second, we define the operator $pre_\rho(Y)$ that gives the set S of classes in ρ such that for each class $W \in S$, there is a state in W that can go to a state in Y. Suppose there is a transition $t = (n_1, c, a, h, n_2)$. Any valuation v after the transition t should satisfy the condition: $\forall x, y \in ivar(a), v(x) = v(y)$. Let Z be the image via t, that is,

$$Z = Z_2 \cap \{v | \forall x, y \in ivar(a). v(x) = v(y)\}.$$

For a valuation v in Z, let v' be a previous valuation of v before executing t. Then, v' must satisfy the following three conditions: 1) for x in $ivar(a)$, $v'(x)$ may be any value in the domain of x since $v(x)$ depends not on $v'(x)$ but on the incoming message; 2) for x not in $ivar(a)$, $v'(x)$ is equal to $h^{-1}(v(x))$; and 3) v' must satisfy condition c. The set of previous valuations of Z_2 before t is defined by:

$$f^{-1}(Z_2, t) = \ c \ \cap \{ \ v \mid \forall x \in (V - ivar(a)). \ v(x) \in h^{-1}(Z)\}.$$

That is, $(n_1, Z_1 \cap f^{-1}(Z_2, t))$ is the set of states in X which can lead to Y via t. The operator pre_ρ is defined as follows:

$$pre_\rho((n_2, Z_2)) = \{(n_1, Z_1) \in \rho | \exists t \text{ from } n_1 \text{ to } n_2 \in T. Z_1 \cap f^{-1}(Z_2, t) \neq \emptyset\}$$

Third, we define the operator $split(X, \rho)$. Suppose there is a transition $t = (n_1, c, a, h, n_2)$. With the transition t, all states in $X_1 = (n_1, Z_1 \cap f^{-1}(Z_2, t))$ can go to Y, whereas no states in $X_2 = X - (n_1, Z_1 \cap f^{-1}(Z_2, t))$ can go to Y. If either $X_1 = \emptyset$ or $X_2 = \emptyset$, then define $Split(X, Y, t)$ to be $\{X\}$. Otherwise, define $Split(X, Y, t)$ to be $\{X_1, X_2\}$. If $Split(X, Y, t)$ is equal to $\{X\}$, then either all or none of the states in X can execute t.

Using the definition of $Split$, we define the operator $split(X, \rho)$ as follows:

function $split((n_1, Z_1), \rho)$
  $\pi := \{(n_1, Z_1)\}$
  for every outgoing transition t from n_1 in T **do**
    Let the target of t be n_2;
    for every $Y = (n_2, *) \in \rho$ **do**
      $\pi := \bigcup_{X' \in \pi} Split(X', Y, t)$;
    end for
  end for
  return π
end function

Termination. Although the state minimization algorithm may not always terminate, each of the following three sets of sufficient conditions on the syntax of CSMs insure termination: 1) CSMs with finite data space. 2) CSMs with data-independent controls, that is, no enabling condition. 3) CSMs with the form of assignments $x := i$ where $i \in dom(x)$.

4.2 An Example

Recall that Figure 1 shows a CSM with two integer variables x, y whose initial values are between 0 and 10. Since x, y are integer variables, $dom(e)$, $dom(f)$ and $dom(g)$ are also the set $\mathcal{N}$ of integers. We apply the algorithm shown in Figure 2 to this example. Let D denote the data space $\mathcal{N} \times \mathcal{N}$. Remember that R is the set of explored reachable classes and R_s is the set of explored stable classes. Each step represents the execution of the while-loop body.

Step 0: Initially, there are three classes, (n_0, D), (n_1, D) and (n_2, D) as shown in Figure 3(a). Furthermore, R_s is empty and R is $\{(n_0, D)\}$, since the set of initial states, $SI = (n_0, 0 < x < 10 \wedge 0 < y < 10)$, is a subset of (n_0, D).

Step 1: We start with (n_0, D). Since there exists an outgoing transition t_1 from n_0,

$$\begin{aligned} split((n_0, D), \rho) &= Split((n_0, D), (n_1, D), t_1) \\ &= \{(n_0, x > 0), (n_0, x \leq 0)\} \end{aligned}$$

by $f^{-1}(D, t_1) = (x > 0)$.

The new class $(n_0, x > 0)$ includes the initial states SI, but $(n_0, x \leq 0)$ does not. So, we have:

$R = \{(n_0, x > 0)\}$, $R_s = \emptyset$, and

$\rho = \{(n_0, x > 0), (n_0, x \leq 0), (n_1, D), (n_2, D)\}$.

Step 2 : Choose the class $(n_0, x > 0)$ from $R - R_s$,

$split((n_0, x > 0), \rho) = \{(n_0, x > 0)\}$,

$post_\rho((n_0, x > 0)) = \{(n_1, D)\}$ since $f(x > 0, t_1) = (x > 1)$

In this step, ρ is not changed and we have:

$R = \{(n_0, x > 0), (n_1, D)\}$, $R_s = \{(n_0, x > 0)\}$,

as shown in Figure 3(b).

Step 3: Let us select $X = (n_1, D)$ in $(R - R_s)$. Then, a state in X may go to $(n_0, x > 0)$ or $(n_0, x \leq 0)$ through the transition t_2, or it may go to (n_2, D) through t_3. Thus, $split((n_1, D), \rho)$ is computed as follows: In the first iteration with t_2 and $(n_0, x > 0)$, since $f^{-1}(x > 0, t_2) = (x > 0 \wedge y > 0)$, we have

$\pi = Split((n_1, D), (n_0, x > 0), t_2) = \{(n_1, x > 0 \wedge y > 0), (n_1, x \leq 0 \vee y \leq 0)\}$.

In the second iteration with t_2 and $(n_0, x \leq 0)$, since $f^{-1}(x \leq 0, t_2) = (x \leq 0 \wedge y >$

0), we have

$Split((n_1, x > 0 \wedge y > 0), (n_0, x \leq 0), t_2) = \{(n_1, x > 0 \wedge y > 0)\}$

$Split((n_1, x \leq 0 \vee y \leq 0), (n_0, x \leq 0), t_2) = \{(n_1, x \leq 0 \wedge y > 0), (n_1, y \leq 0)\}$.

Then, π becomes $\{(n_1, x > 0 \wedge y > 0), (n_1, x \leq 0 \wedge y > 0), (n_1, y \leq 0)\}$. In the last iteration with t_3 and (n_2, D), since $f^{-1}(D, t_3) = (x < 3)$, we get

$Split((n_1, x > 0 \wedge y > 0), (n_2, D), t_3) = \{(n_1, 0 < x < 3 \wedge y > 0), (n_1, x \geq 3 \wedge y > 0)\}$,

$Split((n_1, x \leq 0 \wedge y > 0), (n_2, D), t_3) = \{(n_1, x \leq 0 \wedge y > 0)\}$,

$Split((n_1, y \leq 0), (n_2, D), t_3) = \{(n_1, x < 3 \wedge y \leq 0), (n_1, x \geq 3 \wedge y \leq 0)\}$.

So, X is split into five classes:

$split(X, \rho) = \{(n_1, 0 < x < 3 \wedge y > 0), (n_1, x \geq 3 \wedge y > 0), (n_1, x \leq 0 \wedge y > 0), (n_1, x < 3 \wedge y \leq 0), (n_1, x \geq 3 \wedge y \leq 0)\}$.

Since X is split and $pre_\rho(X) = (n_0, x > 0)$, we now have

$R = \{(n_0, x > 0)\}$, $R_s = \emptyset$, and

$\rho = \{(n_0, x > 0), (n_0, x \leq 0), (n_1, 0 < x < 3 \wedge y > 0), (n_1, x \geq 3 \wedge y > 0), (n_1, x \leq 0 \wedge y > 0), (n_1, x < 3 \wedge y \leq 0), (n_1, x \geq 3 \wedge y \leq 0), (n_2, D)\}$

as shown in Figure 3(c).

Step 4: Since $post_\rho((n_0, x > 0))$ has split, we reconsider the class $(n_0, x > 0)$. It is split into two subclasses $(n_0, x \geq 2)$ and $(n_0, x = 1)$, all of which include the initial states SI. That is,

$R = \{(n_0, x \geq 2), (n_0, x = 1)\}$, $R_s = \emptyset$ and

$\rho = \{(n_0, x \geq 2), (n_0, x = 1), (n_0, x \leq 0), (n_1, 0 < x < 3 \wedge y > 0), (n_1, x \geq 3 \wedge y > 0), (n_1, x \leq 0 \wedge y > 0), (n_1, x < 3 \wedge y \leq 0), (n_1, x \geq 3 \wedge y \leq 0), (n_2, D)\}$

Step 5,6: When we perform $split((n_0, x \geq 2), \rho)$, the class is not split. Similarly, the class $(n_0, x = 1)$ is not split. Thus,

$R_s = \{(n_0, x \geq 2), (n_0, x = 1)\}$

Since

$post_\rho((n_0, x \leq 2)) = \{(n_1, x \geq 3 \wedge y > 0), (n_1, x \geq 3 \wedge y \leq 0)\}$ and

$post_\rho((n_0, x = 1)) = \{(n_1, 0 < x < 3 \wedge y > 0), (n_1, x < 3 \wedge y \leq 0)\}$,

we have $R = \{(n_0, x \geq 2), (n_0, x = 1), (n_1, x \geq 3 \wedge y > 0), (n_1, 0 < x < 3 \wedge y > 0), (n_1, x \geq 3 \wedge y \leq 0), (n_1, x < 3 \wedge y \leq 0)\}$

as shown in Figure 3(d).

Step 7: Let us select a class $X = (n_1, x \geq 3 \wedge y > 0)$ among $(R - R_s)$. Since $split(X, \rho) = \{X\}$, it is stable with respect to ρ. That is, R_s becomes $\{(n_0, x \geq 2), (n_0, x = 1), (n_1, x \geq 3 \wedge y > 0)\}$ without changing R and ρ.

Step 8: Considering $X = (n_1, 0 < x < 3 \wedge y > 0)$,
$split(X, \rho) = \{(n_1, x = 1 \wedge y > 0), (n_1, x = 2 \wedge y > 0)\}$ and $pre_\rho(X) = \{(n_0, x = 1)\}$.
As shown in Figure 3(e), we have:
$R = \{(n_0, x \geq 2), (n_0, x = 1), (n_1, x \geq 3 \wedge y > 0), (n_1, x \geq 3 \wedge y \leq 0)\}$,
$R_s = \{(n_0, x \geq 2), (n_1, x \geq 3 \wedge y > 0)\}$,
$\rho = \{(n_0, x \geq 2), (n_0, x = 1), (n_0, x \leq 0), (n_1, x \geq 3 \wedge y > 0), (n_1, x = 1 \wedge y > 0), (n_1, x = 2 \wedge y > 0), (n_1, x \leq 0 \wedge y > 0), (n_1, x \geq 3 \wedge y \leq 0), (n_1, x < 3 \wedge y \leq 0), (n_2, D)\}$.

Step 9: Considering $(n_0, x = 1)$, it is not split and is added into R_s. And $post_\rho((n_0, x = 1))$ should be added into R. Figure 3(f) shows the current situation such that
$R = \{(n_0, x \geq 2), (n_0, x = 1), (n_1, x \geq 3 \wedge y > 0), (n_1, x \geq 3 \wedge y \leq 0), (n_1, x < 3 \wedge y \leq 0), (n_1, x = 2 \wedge y > 0)\}$,
$R_s = \{(n_0, x \geq 2), (n_0, x = 1), (n_1, x \geq 3 \wedge y > 0)\}$

Step 10: For $(n_1, x < 3 \wedge y \leq 0)$, it is stable with respect to the current partition. And (n_2, D) is in $post_\rho((n_1, x < 3 \wedge y \leq 0)$. Then
$R = \{(n_0, x \geq 2), (n_0, x = 1), (n_1, x \geq 3 \wedge y > 0), (n_1, x \geq 3 \wedge y \leq 0), (n_1, x < 3 \wedge y \leq 0), (n_1, x = 2 \wedge y > 0), (n_2, D)\}$,
$R_s = \{(n_0, x \geq 2), (n_0, x = 1), (n_1, x \geq 3 \wedge y > 0), (n_1, x < 3 \wedge y \leq 0)\}$.

After we consider all the reachable classes in $R - R_s$, we find that they are all stable with respect to the current partition (i.e., $R = R_s$) as shown in Figure 3(g). Therefore, we get the minimal transition system shown in Figure 4.

5 An Application: Minimization of an Ada Program

We apply our approach to a producer-consumer program written in Ada to show its potential in the analysis of concurrent programs.

5.1 A Producer-Consumer Example

Figure 5 describes a producer and a consumer that communicate through a two-slot buffer. There are three tasks: PRODUCER, CONSUMER and BUFFER. The PRODUCER task gets an input item and sends it to the BUFFER task. The PRODUCER task is forced to wait if the BUFFER task holds two items The CONSUMER task receives an item from the BUFFER task and outputs it. The CONSUMER task is forced to wait if the BUFFER task does not hold any item.

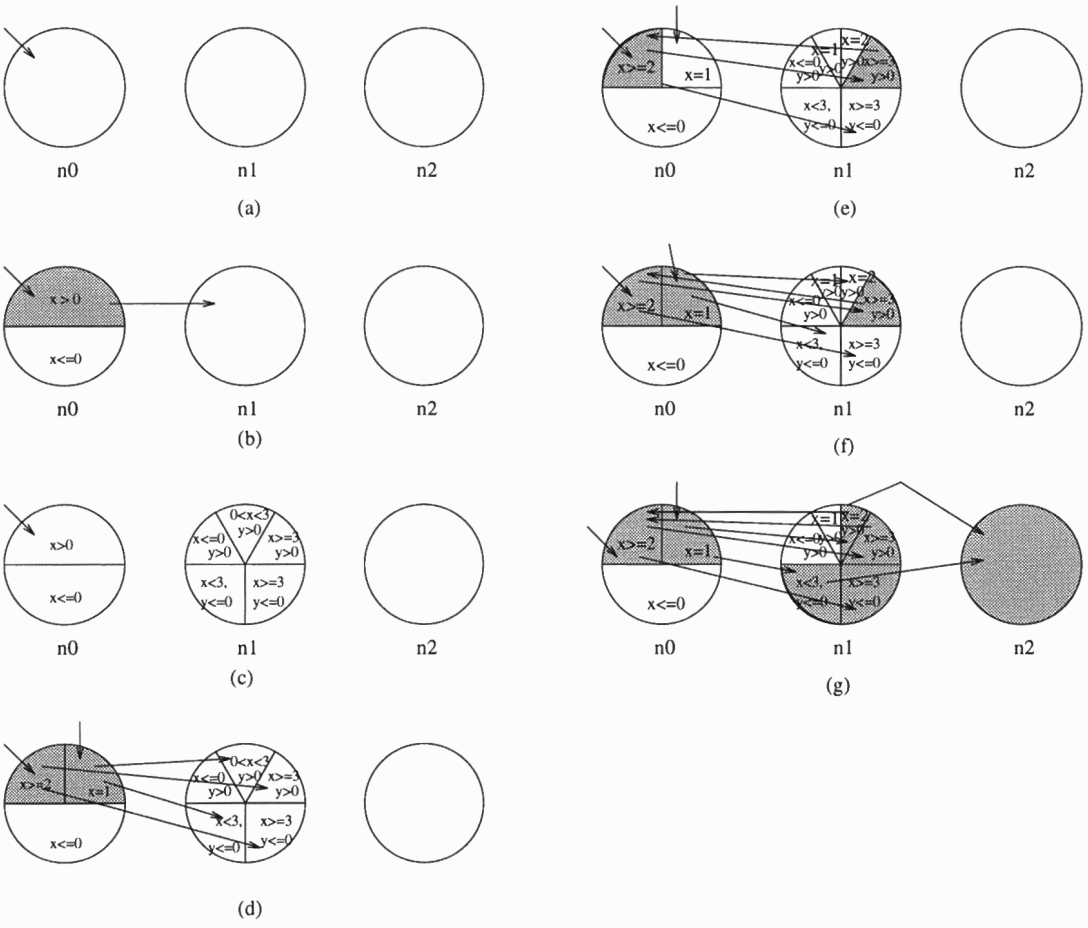


Figure 3: Split steps for the simple CSM example

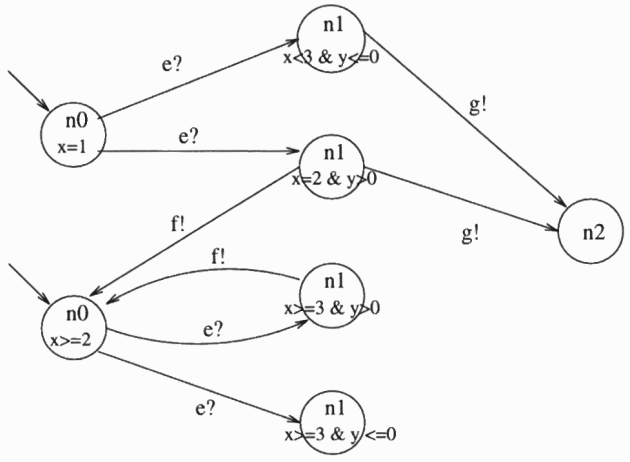


Figure 4: A Minimal Transition System of the Simple CSM

```
task PRODUCER;
task CONSUMER;
task BUFFER is
  entry WRITE(Z: in INTEGER);          % channels: s_WRITE and f_WRITE
  entry READ(Z: out INTEGER);          % channels: s_READ and f_WRITE
end BUFFER;

task body PRODUCER is
  X: INTEGER;
  begin                                % make the initial node 1
    loop                               %
      GET(X);                          % put an edge (1, true,GET?(X), ∅, 2)
      BUFFER.WRITE(X);                 % put an edge (2, true,s_WRITE!(X), ∅, 3)
                                       % and an edge (3, true,f_WRITE?, ∅, 3')
    end loop                           % replace 3' to 1
  end PRODUCER

task body CONSUMER is
  Y: INTEGER;
  begin                                % make the initial node 4
    loop                               %
      BUFFER.READ(Y);                  % put an edge (4, true,s_READ!, ∅, 5)
                                       % and an edge (5, true,f_READ?(Y), ∅, 6)
      PUT(Y);                          % put an edge (6, true,PUT!(Y), ∅, 6')
    end loop                           % replace 6' to 4
  end CONSUMER

task body BUFFER is
  N: constant INTEGER := 3;            % initialize dom(N) = 𝒩 and I(N) = 3
  Q: array(1..N) of INTEGER;           % Initialize dom(Q(1)),...dom(Q(3)) = 𝒩
  INB, OUTB: INTEGER := 1;             % Initialize dom(INB), dom(OUTB) = 𝒩, I(INB), I(OUTB) = 1
  begin                                % make the initial node 7
    loop                               %
      select
        when INB mod N + 1 ≠ OUTB ⇒
          accept WRITE(Z: in INTEGER) do   % put an edge (7, INB mod N + 1 ≠ OUTB,s_WRITE?(Z), ∅, 8)
            Q(INB mod N + 1) := Z;         % put an edge (8, true, , {Q(INB mod N + 1) := Z}, 9)
          end WRITE                        % put an edge (9, true,f_WRITE!, ∅, 10)
          INB := INB mod N + 1;            % put an edge (10, true, , {INB := INB mod N + 1}, 10')
        or when INB ≠ OUTB ⇒
          accept READ(Z: out INTEGER) do   % put an edge (7, INB ≠ OUTB,s_READ?, ∅11)
            Z := Q(OUTB mod N + 1);        % put an edge (11, true, , {Z := Q(OUTB mod N + 1)}, 12)
          end READ                         % put an edge (12, true,f_READ!(Z), ∅, 13)
          OUTB := OUTB mod N + 1;          % put an edge (13, true, , {OUTB := OUTB mod N + 1}, 13')
      end select                           % merge 10' and 13'
    end loop                               % replace the merged vertex into 7
  end BUFFER
```

Figure 5: A Producer-Consumer program in Ada

The three tasks PRODUCER, CONSUMER and BUFFER can be translated to CSMs as shown in Figure 5. The translation is straightforward except for Ada's rendezvous construct. We simulate Ada's rendezvous with two actions: one for start and another for end.

- "**entry** WRITE(Z:**in** INTEGER)" of BUFFER creates two channels: s_WRITE and f_WRITE, where *dom*(s_WRITE) is the set of integers. In PRODUCER, the statement "BUFFER.WRITE(X)" is translated into two actions, s_WRITE!(X) and f_WRITE?, for starting and finishing rendezvous through the entry WRITE of BUFFER. In BUFFER, the statement "accept WRITE(Z: **in** integer)" is translated into s_WRITE?$\{Z\}$ and the statement "end WRITE" into f_WRITE! to synchronize and communicate with PRODUCER.

- "**entry** READ(Z:**out** INTEGER)" of BUFFER creates two channels: s_READ and f_READ, where *dom*(f_READ) is the set of integers. In CONSUMER, "BUFFER.READ(Y)" becomes s_READ! and f_READ?$\{Y\}$. In BUFFER, "accept READ(Z: **out** integer)" and "end READ" is translated into s_READ? and f_READ!(Z), respectively.

The difference between WRITE and READ is that PRODUCER sends a message to BUFFER through s_WRITE at the start time of WRITE rendezvous, but CONSUMER receives a message from BUFFER through f_READ at the end time of READ rendezvous. The resulting CSMs are given in Figure 6.

We compose the three CSMs and construct the global CSM as shown in Figure 7. The CSM has infinitely many states since it includes integer variables. Thus, it is not possible to enumerate and analyze all reachable states directly. When we apply our approach to this CSM, we obtain the finite and minimal reachable transition system shown in Figure 8.

5.2 Analysis of the Producer-Consumer Program

Let $\#(e, b)$ denote the number of occurrences of event e within behavior b. To be correct, the following properties should be satisfied by the Producer-Consumer program.

1. The number of messages written into BUFFER is greater than or equal to the number of messages read from BUFFER and is less than or equal to the number of messages read from BUFFER + 2. That is, for every behavior b,

$$\#(\text{s\_READ}, b) \leq \#(\text{s\_WRITE}, b) \text{ and } \#(\text{s\_WRITE}, b) \leq \#(\text{s\_READ}, b) + 2$$

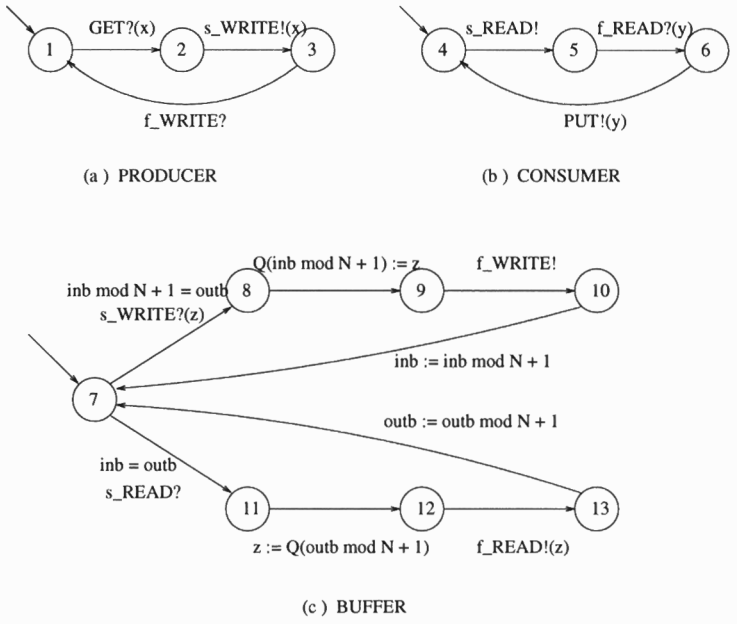


Figure 6: Producer, Consumer and Buffer CSMs

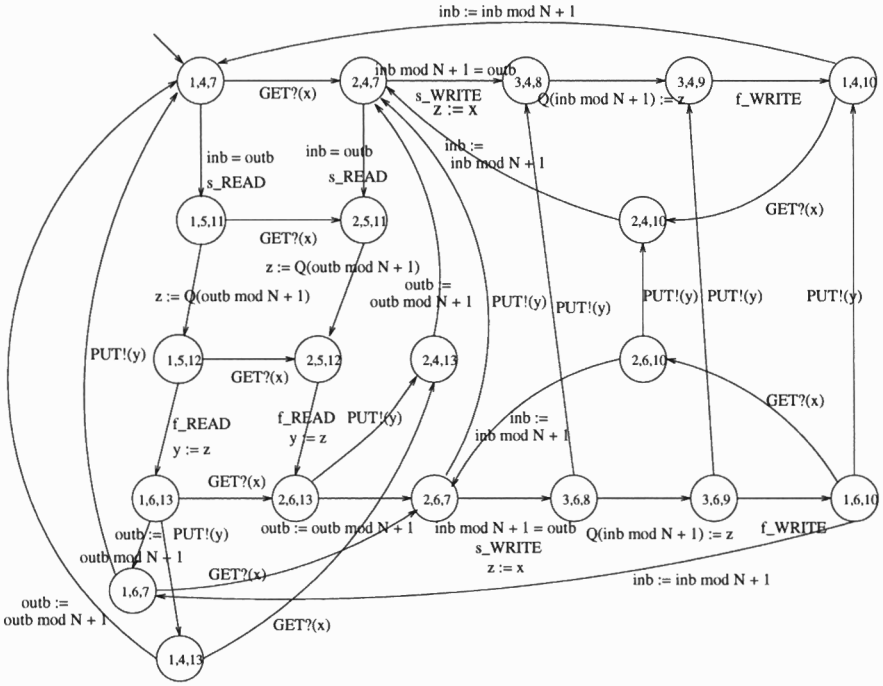


Figure 7: A CSM corresponding to the Producer-Consumer program

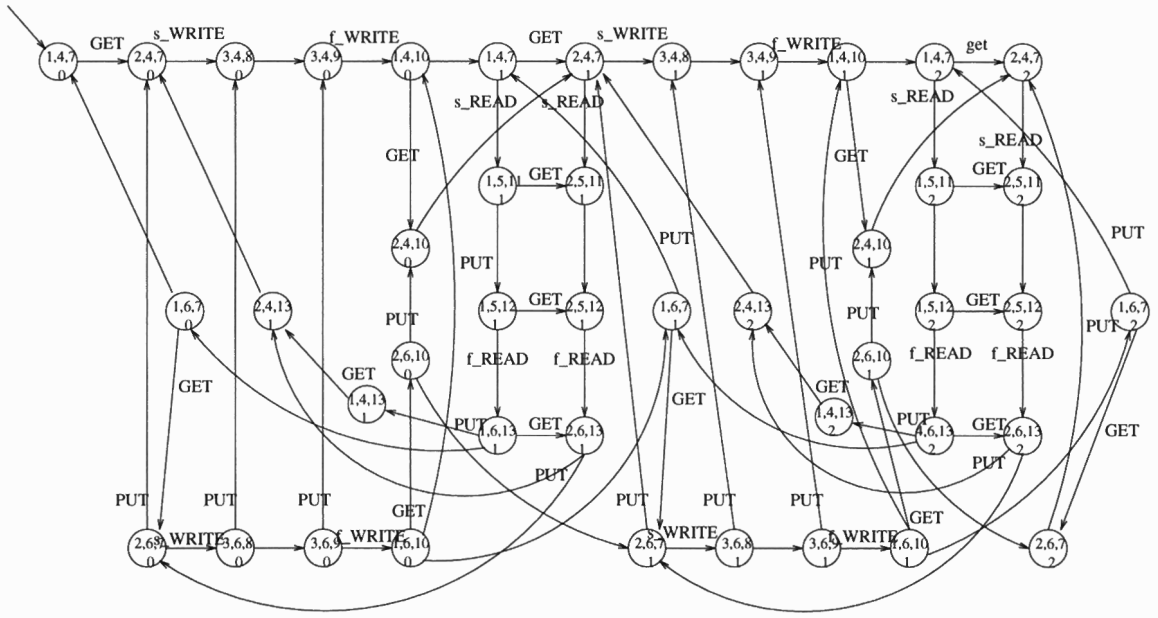


Figure 8: A minimal transition system of the Producer-Consumer program

2. The number of messages generated from PRODUCER is greater than or equal to the number of messages consumed from CONSUMER. That is, for every behavior b,

$$\#(\text{GET}, b) \geq \#(\text{PUT}, b)$$

We give an algorithm in Figure 9 to decide whether the minimal transition system in Figure 8 satisfies the first property. For an element (v, i), v represents a vertex in the transition system and i represents

$$\#(\text{s\_WRITE}, b) - \#(\text{s\_READ}, b),$$

where b is a behavior before entering the vertex. Since the possible values of i are 0, 1, and 2, the execution time of the above algorithm is proportional to the size of the transition system. It can be shown that the Producer-Consumer program satisfies the first property using the decision algorithm. The second property can also be shown using the decision algorithm modified to keep track of the numbers of GETs and PUTs.

6 Conclusion

We have presented an algorithm to overcome the state explosion problem of reachability analysis caused by large data space size. Our algorithm computes a minimal reachable state space of a system described as communicating state machines with infinite data

```
algorithm Decision for Property 1

    Unexplored := { (v, 0) | v is an initial vertex in T};
    Explored := ∅;
    while Unexplored is not empty do
        select a vertex (v, i) in Unexplored;
        add (v, i) into Explored;
        for every edge e from v to a vertex (say v') in T do
            if the edge is labeled with s_READ then
                i' := i − 1;
                if i' < 0 then
                    report false and terminate;      % #(s_READ, b) < #(s_WRITE, b)
            else if the edge is labeled with s_WRITE then
                i' := i + 1;
                if i' > 2 then
                    report false and terminate;      % #(s_WRITE, b) > #(s_READ, b) + 2
            else i' := i;

            if (v', i') is not in Explored then
                add (v', i') into Unexplored;
        end for
    end while
    report true;              % #(s_READ, b) ≤ #(s_WRITE, b) ≤ #(s_READ, b) + 2
end algorithm
```

Figure 9: Decision Algorithm

space. We use the notion of bisimulation as the underlying equivalence for state minimization. The salient aspect of our approach is that a minimal state space is constructed without explicitly generating the entire state space. This is very important because it is not possible to generate all the reachable states of a system if the number of states is very large or infinite. Our algorithm extends the algorithm of Bouajjani *et al.* [2] to a labeled transition system and also allows infinitely many initial states. Since the algorithm may not terminate, we have identified the set of sufficient conditions on the syntax of communicating state machines that guarantee termination. To illustrate the usefulness of our approach, we have illustrated how to translate the producer-consumer program written in Ada to communicating state machines, how to minimize the state space of the resultant communicating state machines, and how to check for its correctness.

There are several areas that we are currently working on: First, we are currently investigating other sets of sufficient conditions for guaranteeing termination. Second, we are investigating how to do model checking using a minimal state space generated by our algorithm for properties written in trace logic used with the producer-consumer example. Third, we are implementing the minimization algorithm as part of the toolkit called VERSA [3]. This would allow us to experimentally evaluate its effectiveness. Fourth, we are developing minimization techniques based on notions of equivalence other than bisimulation. Fifth, we have extended communicating state machines with time and probability and is currently investigating how to generate a minimal state space from such machines.

References

[1] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of Timed Transition Systems. In W.R. Cleaveland, editor, *Proceedings of International Conference on Concurrency Theory*, Lecture Notes in Computer Science vol. 630. Springer-Verlag, August 1992.

[2] A. Bouajjani, J.-C. Fernandez, and N. Halbwachs. Minimal Model Generation. In *Proceedings of Workshop on Computer-Aided Verification*, 1990.

[3] Duncan Clarke, Insup Lee, and Hong liang Xie. VERSA: A Tool for the Specification and Analysis of Resource-Bound Real-Time Systems. Technical Report MS-CIS-93-77, Dept. of CIS, Univ. of Pennsylvania, Sept 1993.

[4] 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, April 1986.

[5] E. M. Clarke, D. E. Long, and K. L. McMillan. Compositional Model Checking. In *Proceedings of Fourth Annual Symposium on Logic in Computer Science*, 1989.

[6] J.-C. Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 13:219–236, 1990.

[7] B. Jonsson and J. Parrow. Deciding Bisimulation Equivalences for a Class of Non-finite-state Programs. Technical Report SICS/R-89/8908, Swedish Institute of Computer Science, August 1989.

[8] P. C. Kanellakis and S. A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Information and Computation*, 86:43–68, 1990.

[9] Inhye Kang and Insup Lee. A State Minimization Algorithm for Communicating State Machines with Arbitrary Data Space. Technical Report MS-CIS 93-07, Dept. of Computer and Information Science, Univ. of Pennsylvania, 1993.

[10] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[11] R. Paige and R.E. Tarjan. Three Partition Refinement Algorithms. *SIAM J. Comput.*, 16(6), December 1987.

[12] A. C. Shaw. Communicating Real-Time State Machines. Technical Report 91-08-09, Dept. of Computer Science and Engineering, Univ. of Washington, 1991.

[13] W. J. Yeh and M. Young. Compositional Reachability Analysis using Process Algebra. In *Proceedings of Conference on Testing, Analysis and Verification*, August 1991.