Model Checking for Programming Languages using VeriSoft

Patrice Godefroid
Bell Laboratories
Lucent Technologies
1000 E. Warrenville Road
Naperville, IL 60566, U.S.A.
god@bell-labs.com
http://www.bell-labs.com/people/god

Abstract

Verification by state-space exploration, also often referred to as "model checking", is an effective method for analyzing the correctness of concurrent reactive systems (e.g., communication protocols). Unfortunately, existing model-checking techniques are restricted to the verification of properties of models, i.e., *abstractions*, of concurrent systems.

In this paper, we discuss how model checking can be extended to deal directly with "actual" descriptions of concurrent systems, e.g., *implementations* of communication protocols written in programming languages such as C or C++. We then introduce a new search technique that is suitable for exploring the state spaces of such systems. This algorithm has been implemented in *VeriSoft*, a tool for systematically exploring the state spaces of systems composed of several concurrent processes executing arbitrary C code. As an example of application, we describe how VeriSoft successfully discovered an error in a 2500-line C program controlling robots operating in an unpredictable environment.

1 Introduction

Concurrent systems are systems composed of elements that can operate concurrently and communicate with each other. Each component can be viewed as a *reactive system*, i.e., a system that continuously interacts with its environment. Concurrent reactive systems are notably hard to design because their components may interact in many unexpected ways. Traditional testing techniques are of limited help since test coverage is bound to be only a minute fraction of the possible behaviors of the system.

State-space exploration is one of the most successful strategies for analyzing the correctness of concurrent reactive systems. It consists of exploring a directed graph, called the *state space*, representing the combined behavior of all concurrent components in a system. Such a state space can be computed automatically from a description of the concurrent system specified in a *modeling language*. Many properties of a model of a system can be checked by exploring its state space: deadlocks, dead code, violations of user-specified assertions, etc. Moreover, the range of properties that state-space exploration techniques can verify has been substantially broadened during the last decade thanks to the development of model-checking methods for various temporal logics (e.g., [CES86, LP85, QS81, VW86]). In what follows, we will use the term "model checking" in a broad sense, to denote any automatic state-space exploration technique that can be used for verification purposes.[1]

Examples of tools that follow the above paradigm are CAESAR [FGM$^+$92], COSPAN [HK90], CWB [CPS93], MURPHI [DDHY92], SMV [McM93], SPIN [Hol91], and VFSMvalid [FHS95], among others. These tools differ by the modeling languages they use for representing systems and properties, and by the conformation criteria according to which these representations are compared. But all of them are based on state-space exploration algorithms, in one form or another, for performing the verification itself.

The effectiveness of model checking for debugging concurrent reactive systems is becoming increasingly well-established. Several very complex concurrent systems have been modeled, and then analyzed using state-

[1] Note that the term "model checking" is not due to the fact that the correctness of a model, i.e., an abstraction, of a system is checked, but rather refers to the fact that model checking checks whether all the computations of a system are "models", in the classical logical sense, of a temporal logic formula.

space exploration techniques. In many cases, these techniques were able to reveal quite subtle design errors (e.g., [Rud92, CGH+93, BG96]).

It is worth emphasizing that the practical interest of these state-space exploration techniques (and of "verification" in general) is mainly to find errors that would be hard to detect and reproduce otherwise, and not necessarily to prove the absence of errors. While mathematically proving that a model of a system conforms to a specific set of properties does increase the confidence that the actual system is "correct", it does not provide a proof of this fact.

In this paper, we discuss how model checking can be extended to deal directly with "actual" descriptions of concurrent systems, e.g., *implementations* of communication protocols written in programming languages such as C or C++. We show that existing search techniques for state-space exploration are fundamentally limited to the analysis of systems for which each state of the system can be readily represented by a unique identifier. We then introduce an efficient search technique that does not rely on this assumption. This search algorithm can therefore be applied to systems composed of several concurrent processes executing arbitrary code written in full-fledged programming languages.

2 Concurrent Systems and Dynamic Semantics

We consider a concurrent system composed of a finite set $\mathcal{P}$ of *processes* and a finite set $\mathcal{O}$ of *communication objects*. Each process $P \in \mathcal{P}$ executes a sequence of *operations*, that is described in a sequential program written in a full-fledged programming language such as C or C++. Such programs are *deterministic*: every execution of the program on the same data performs the same sequence of operations. We assume that processes communicate with each other by performing operations on communication objects. A communication object $O \in \mathcal{O}$ is defined by a pair (V, OP), where V is the set of all possible values for the object (its domain), and OP is the set of *operations* that can be performed on the object. Examples of communication objects are shared variables, semaphores, and FIFO buffers. At any time, at most one operation can be performed on a given communication object (operations on a same communication object are mutually exclusive). Operations on communication objects are called *visible operations*, while other operations are by default called *invisible*. The execution of an operation is said to be *blocking* if it cannot be completed. We assume that only executions of visible operations may be blocking.

At any time, the concurrent system is said to be in a *state*. The system is said to be in a *global state* when the next operation to be executed by every process in the system is a visible operation. Initially, after the creation of all the processes of the system, we assume that all the processes eventually executes a visible operation, and hence that the system may reach a first and unique global state s_0, called the *initial global state* of the system. We define a *process transition*, or *transition* for short, as one visible operation followed by a *finite* sequence of invisible operations performed by a single process. Let $\mathcal{T}$ denote the set of all transitions of the system.

A transition is said to be *disabled* in a global state s when the execution of its visible operation is blocking in s. Otherwise, the transition is said to be *enabled* in s. A transition t that is enabled in a global state s can be *executed* from s. Since the number of invisible operations in a transition is finite, the execution of an enabled transition always terminates. When the execution of t from s is completed, the system reaches a global state s', called the *successor* of s by t.[2] We write $s \xrightarrow{t} s'$ to mean that the execution of the transition t leads from the global state s to the global state s', while $s \overset{w}{\Rightarrow} s'$ means that the execution of the finite sequence w of transitions leads from s to s'. If $s \overset{w}{\Rightarrow} s'$, s' is said to be *reachable* from s.

We now define a formal semantics for the concurrent systems that satisfy our assumptions. A concurrent system as defined here is a closed system: from its initial global state, it can evolve and change its state by executing enabled transitions. Therefore, a very natural way to describe the possible *behaviors* of such a system is to consider its set of reachable global states and the transitions that are possible between these.

Formally, the joint *global* behavior of all processes P_i in a concurrent system can be represented by a transition system $A_G = (S, \Delta, s_0)$ such that

- S is the set of global states of the system,
- $\Delta \subseteq S \times S$ is the *transition relation* defined as follows:

$$(s, s') \in \Delta \text{ iff } \exists t \in \mathcal{T} : s \xrightarrow{t} s',$$

- s_0 is the initial global state of the system.

An element of Δ corresponds to the execution of a single transition $t \in \mathcal{T}$ of the system. The elements of Δ will be referred to as *global transitions*. It is natural to restrict A_G to its global states and transitions that are reachable from s_0, since the other global states and

[2] Operations on objects (and hence transitions) are deterministic: the execution of a transition t in a state s leads to a *unique* successor state.

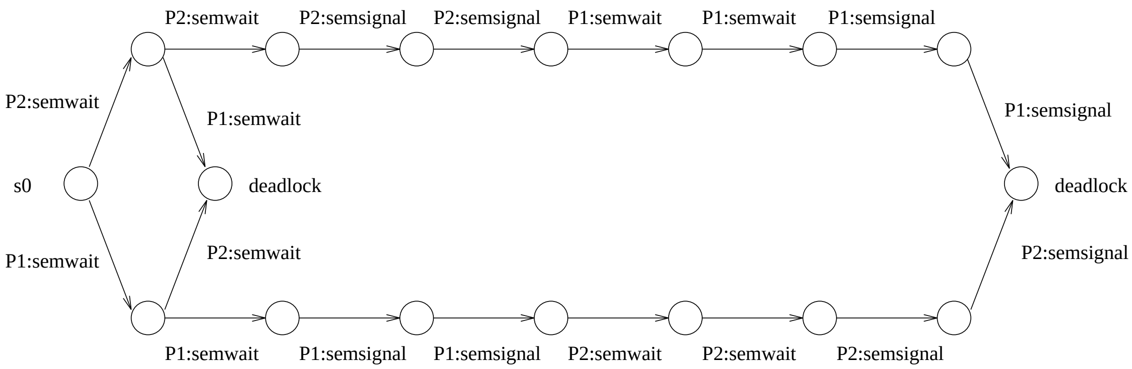

Figure 1: Global state space for the two-dining-philosophers system

transitions play no role in the behavior of the system. In what follows, a "state in A_G" denotes a state that is reachable from s_0. By definition, states in A_G are global. A_G is called the *global state space* of the system.

Example 1 Consider the following concurrent C program.

```
/* phil.c : dining philosophers (version without loops) */

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define N 2

philosopher(i)
     int i;
{
  printf("philosopher %d thinks\n",i);
  semwait(i);           /* take left fork */
  semwait((i+1)%N);     /* take right fork */
  printf("philosopher %d eats\n",i);
  semsignal(i);         /* release left fork */
  semsignal((i+1)%N);   /* release right fork */
  exit(0);
}

main()
{
  int semid, i, pid;

  semid = semget(IPC_PRIVATE,N,0600);

  for(i=0;i<N;i++)
    semsetval(i,1);
  for(i=0;i<(N-1);i++) {
    if((pid=fork()) == 0)
      philosopher(i);
    };
  philosopher(i);
}
```

This program represents a concurrent system composed of two processes. It describes the behavior of these processes as well as the initialization of the system. This example is inspired by the well-known dining-philosophers problem, with two philosophers. The two processes communicate by executing the (visible) operations *semwait* and *semsignal* on two semaphores that are identified by the integers 0 and 1 respectively. The value of both semaphores is initialized to 1 (with the operation *semsetval*). By implementing these operations using actual UNIX semaphores, the program above can be compiled and run on any UNIX machine. The state space A_G of this system is shown in Figure 1, where global transitions are labeled with the visible operation of the corresponding process transition. The operation *exit* is a visible operation whose execution is always blocking. Since all the processes are deterministic, nondeterminism in A_G is caused only by concurrency. ■

Since we consider here closed concurrent systems, the environment of one process is formed by the other processes in the system. This implies that, in the case of a single "open" reactive system, the environment in which this system operates has to be represented, possibly using other processes. In practice, a complete representation of such an environment may not be available, or may be very complex. It is then convenient to use a *model*, i.e., a simplified representation, of the environment to simulate its external behavior. For this purpose, we introduce a special operation "VS_toss" to express a valuable feature of modeling languages, not found in programming languages: *nondeterminism*. This operation takes as argument a positive integer n, and returns an integer in $[0, n]$. The operation is visible and nondeterministic: the execution of a transition starting with VS_toss(n) may yield up to $n + 1$ different successor states, corresponding to different values returned by VS_toss.

Which properties of a concurrent system is it possible to check by examining its state space A_G as defined above? Here, we focus mainly on two verification problems (other properties will be discussed later in Section 5): the detection of *deadlocks*, i.e., states where the execution of the next operation of every process in the system is blocking, and the detection of violations of *assertions* specified by the user with the special operation "VS_assert". This operation can be inserted in the code of any process, and is considered visible. It takes as its argument a boolean expression that can test and compare the value of variables and data structures local to the process. When "VS_assert(expression)" is executed, the expression is evaluated. If the expression evaluates to false, the assertion is said to be *violated*.

The following theorem states that deadlocks and assertion violations can be detected by exploring only the global states of a concurrent system.

Theorem 1 *Consider a concurrent system as defined above, and let A_G denote its state space. Then, all the deadlocks that are reachable after the initialization of the system are global states, and are therefore in A_G. Moreover, if there exists a state reachable after the initialization of the system where an assertion is violated, then there exists a global state in A_G where the same assertion is violated.*

Proof: See Appendix. ■

This theorem justifies our choice for the "dynamic" semantics described in this section.

In the next section, we discuss how to build a representation of the state space of a concurrent system as defined above. We briefly review standard state-space exploration techniques, and show why they are not appropriate for exploring state spaces of concurrent systems whose processes are described by arbitrary programs.

3 Existing State-Space Exploration Techniques

In the case of *models* of concurrent systems, a state space A_G is usually computed by performing a search of all the states that are reachable from the initial state s_0 of the system. An algorithm for performing such a search is shown in Figure 2. This algorithm recursively explores all successor states of all states encountered during the search, starting from the initial state, by executing all enabled transitions in each state (lines 7–8). The main data structures used are a *Set* to store the states whose successors still have to be explored, and a hash table H to store all the states that have already

```
1   Initialize: Set is empty; H is empty;
2               add s0 to Set;
3   Loop: while Set ≠ ∅ do {
4               take s out of Set;
5               if s is NOT already in H then {
6                 enter s in H;
7                 T = enabled(s);
8                 for all t in T do {
9                      s' = succ(s) after t;
10                     add s' to Set;
11                     }
12                }
13              }
```

Figure 2: Algorithm 1 – classical search

been visited during the search. The set of all transitions enabled in a state s is denoted by *enabled*(s). The state reached from a state s after the execution of a transition t is denoted "succ(s) after t". It is easy to prove that, if A_G is finite, all the states of A_G are visited during the search performed by the algorithm of Figure 2 [AHU74]. The order in which the search is performed (e.g., depth-first, breadth-first, ...) depends on how the operations "add" and "take" are implemented.

It is important to note that the algorithm of Figure 2 assumes that each state s can be represented by a *unique identifier*, that can be stored in the data structures *Set* and H during the search. Although other search algorithms for modeling languages , such as symbolic verification methods [BCM$^+$90, CGL92, McM93], may use other types of data structures (e.g., Binary Decision Diagrams [Bry92]) for representing state spaces, they all rely on the assumption that each state of the system has a unique and manageable representation.

When dealing with processes described by arbitrary programs written in full-fledged programming languages, this assumption is not valid anymore. Indeed, the state of each process is determined by the values of all the memory locations that can be accessed by the process and influence its behavior (including activation records associated to procedure calls). This information is typically far too large and complex to be efficiently and unambiguously encoded by a string of bits, which could then be saved in memory at each step of the state-space exploration.

However, nothing prevents us from systematically searching the state space of a concurrent system without storing any intermediate states in memory. Let us call such a search a *state-less search*. Of course, if the state space A_G contains cycles, a state-less search through it will not terminate, even if A_G is finite. Even state-less searches of "small" finite acyclic state spaces (e.g., composed of only a few thousand states) may not terminate in a reasonable amount of time. To illustrate this phenomenon, let us consider the dining-philosophers example again. (The state space of this system does not contain any cycles.) The number of transitions explored by a classical search (Algorithm 1) and by a state-less search are compared in Figure 4, for various numbers N of philosophers. The run-time of both algorithms is proportional to the number of explored transitions. One clearly sees that the state-less search is much slower than the classical one. In the case of four philosophers, the state-less search explores 386816 transitions, while they are only 708 transitions in A_G. While every transition of A_G is executed exactly once during a classical search, every transition of A_G is executed on average about 546 times during a state-less search! This tremendous difference is due to the numerous re-explorations of unstored parts of the state space during the state-less search.

4 An Efficient State-Less Search Algorithm

The state-less search technique can be viewed as a particular case of *state-space caching* [Hol85, JJ91, GHP95], a memory management technique for storing the states encountered during a classical search performed in depth-first order. State-space caching consists of storing all the states of the current explored path plus as many other states as possible given the remaining amount of available memory. It thus creates a restricted *cache* of selected states that have already been visited. This method never tries to store more states than possible in the cache. A state-less search corresponds to the extreme case where the cache does not contain any state at all.

State-space caching suffers the same drawback as the state-less search: multiple redundant explorations of large unstored parts of the state space yield an unacceptable blow-up of the run-time. Indeed, almost all states in the state space of concurrent systems are typically reached several times during the search. There are two causes for this:

1. From the initial state, the exploration of any interleaving of a single finite partial ordering of transitions of the system always leads to the same state. This state will thus be visited several times because of all these interleavings.

2. From the initial state, explorations of different finite partial orderings of transitions may lead to the same state.

In [GHP95], it is shown that most of the effects of the first cause given above can be avoided when using a search algorithm based on the notion of *sleep sets* [God90, GW93]. Such an algorithm dynamically prunes the state space of a concurrent system without incurring the risk of any incompleteness in the verification results. Empirical results [GHP95, God96] show that, in many cases, most of the states are visited *only once* during a state-space exploration performed with this search technique. This makes it possible not to store most of the states previously visited during the search without incurring much redundant exploration of parts of the state space.

Sleep sets belong to a broader family of algorithms, referred to as *partial-order methods* [God96], that were developed to tackle the "state explosion" phenomenon that limits the efficiency and applicability of verification by state-space exploration. In [God96], it is shown that sleep sets can be combined with another pruning technique based on the notion of *persistent sets*. Using both techniques simultaneously preserves the beneficial properties of sleep sets outlined in the previous paragraph while substantially reducing the number of states and transitions that have to be visited.

In this section, we present a new state-space exploration algorithm that combines a state-less search with the persistent-set and sleep-set techniques. Before turning to the presentation of this algorithm, we briefly recall some basic principles of partial-order methods.

The basic idea behind partial-order methods that enables them to check properties of A_G without constructing the whole of A_G is the following: A_G contains many paths that correspond simply to different execution orders of the same process transitions. If these transitions are "independent", for instance because they are executed by noninteracting processes, then changing their order will not modify their combined effect.

This notion of independency between transitions and its complementary notion, the notion of dependency, can be formalized by the following definition (adapted from [KP92]).

Definition 1 Let $\mathcal{T}$ be the set of system transitions and $D \subseteq \mathcal{T} \times \mathcal{T}$ be a binary, reflexive, and symmetric relation. The relation D is a *valid dependency relation* for the system iff for all $t_1, t_2 \in \mathcal{T}$, $(t_1, t_2) \notin D$ (t_1 and t_2 are *independent*) implies that the two following properties hold for all global states s in the global state space A_G of the system:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s' (independent transitions can neither disable nor enable each other); and

2. if t_1 and t_2 are enabled in s, then there is a unique state s' such that $s \stackrel{t_1 t_2}{\Rightarrow} s'$ and $s \stackrel{t_2 t_1}{\Rightarrow} s'$ (commutativity of enabled independent transitions).

■

This definition characterizes the properties of possible "valid" dependency relations for the transitions of a given system. In practice, it is possible to give easily checkable syntactic conditions that are sufficient for transitions to be independent. In a concurrent system as defined in Section 2, dependency can arise between transitions of different processes that refer to the same communication objects. For instance, two wait operations on a binary semaphore are dependent when they are enabled, while two signal operations on the same non-binary semaphore are independent. Carefully tracking dependencies between operations on communication objects is by no means a trivial task. We refer the reader to [God96] for a detailed presentation of that topic.

All partial-order algorithms follow the same basic pattern: they operate as classical state-space searches except that, at each state s reached during the search, they compute a subset T of the set of transitions enabled at s, and explore only the transitions in T, the other enabled transitions are not explored. Such a search is called a *selective search*. It is easy to see that a selective search through A_G only reaches a subset (not necessarily proper) of the states and transitions of A_G.

Two main techniques for computing such sets T have been proposed in the literature: the persistent-set and sleep-set techniques. The first technique actually corresponds to a whole family of algorithms [Ove81, Val91, GP93, GW93, Pel93]. In [God96], it is shown that all these algorithms compute "persistent sets". Intuitively, a subset T of the set of transitions enabled in a state s of A_G is called *persistent in s* if all transitions not in T that are enabled in s, or in a state reachable from s through transitions not in T, are independent with all transitions in T. In other words, whatever one does from s, while remaining outside of T, does not interact with or affect T. Formally, we have the following [GP93].

Definition 2 A set T of transitions enabled in a state s is *persistent in s* iff, for all nonempty sequences of transitions

$$s = s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} s_3 \ldots \xrightarrow{t_{n-1}} s_n \xrightarrow{t_n} s_{n+1}$$

from s in A_G and including only transitions $t_i \notin T$, $1 \leq i \leq n$, t_n is independent with all transitions in T. ■

Note that the set of all enabled transitions in a state s is trivially persistent since nothing is reachable from s by transitions that are not in this set. It is beyond the scope of this paper to present algorithms for computing persistent sets. In a nutshell, these algorithms infer the persistent sets from the static structure of the system

```
1   Initialize: Stack is empty;
2   Search() {
3     DFS(∅);
4     }
5   DFS(set: Sleep) {
6       T = Persistent_Set()\Sleep;
7       while T ≠ ∅ do {
8           take t out of T;
9           push (t) onto Stack;
10          Execute(t);
11          DFS({t' ∈ Sleep | t' and t are independent});
12          pop t from Stack;
13          Undo(t);
14          Sleep = Sleep ∪ {t};
15          };
16      }
```

Figure 3: Algorithm 2 – state-less depth-first search using persistent sets and sleep sets

being verified. They differ by the type of information about the system that they use. The aim of these algorithms is to obtain the smallest possible nonempty persistent sets. See [God96] for several such algorithms and a comparison of their complexity.

The second technique for computing the set of transitions T to consider in a selective search is the sleep set technique [God90, GW93]. This technique does not exploit information about the static structure of the system, but rather about the past of the search. Used in conjunction with a persistent set algorithm, sleep sets can further reduce the number of explored states and transitions.

An algorithm that combines persistent sets and sleep sets with a state-less search is shown in Figure 3. This algorithm performs a selective depth-first search (DFS) in the state space of a concurrent system. The data structure *Stack* contains the sequence of transitions that leads from the initial global state s_0 to the current global state being explored. A set denoted by *Sleep* is associated with each global state reached during the search, i.e., with each call to the procedure DFS. The sleep set associated with a global state s is a set of transitions that are *enabled* in s but *will not be explored* from s. The sleep set associated with the initial global state s_0 is the empty set. Each time a new global state s is encountered during the search, a call to DFS is executed. The sleep set that has to be associated with s is passed as argument. In line 6, a new set of transitions is selected to be explored from s. Persistent_Set() returns a persistent set in the current global state s that is nonempty if there exist transitions enabled in s. Lines 11 and 14 describe how to compute the sleep sets as-

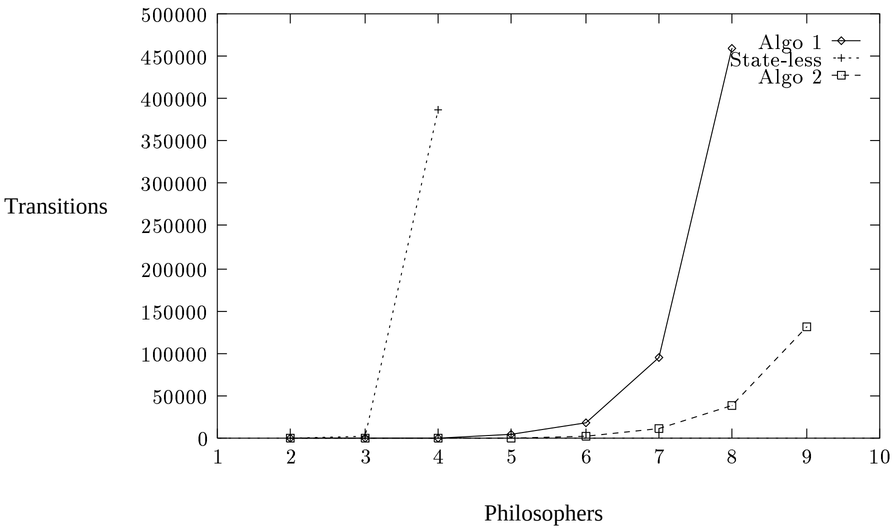

Figure 4: Comparison of performances for the dining-philosophers system

sociated with the successor global states of s from the value of its sleep set *Sleep*. In line 10, a transition t is executed from s. The procedure Execute(t) returns after a new global state has been reached by the concurrent system. Then all the transitions of *Sleep* that are independent with t are passed into the sleep set associated to that new global state (line 11). Once the search from that new state (and hence the corresponding call to DFS) is completed, the exploration of the other transitions selected to be explored from s may proceed. The concurrent system is then brought back to the global state s in line 13. (This can be done by reinitializing the system and reexecuting the sequence of transitions in *Stack*, for instance.) Next, transition t, i.e., the last transition explored from s, is added to *Sleep* in line 14.

The correctness of Algorithm 2 is established by the following theorem.

Theorem 2 *Consider a concurrent system as defined in Section 2, and let A_G denote its state space. Assume A_G is finite and acyclic. Then, all the deadlocks in A_G are visited by Algorithm 2. Moreover, if there exists a global state in A_G where an assertion is violated, then there exists a global state visited by Algorithm 2 where the same assertion is violated.*

Proof: See Appendix. ■

In other words, deadlocks and assertion violations can be detected using Algorithm 2. As discussed in the previous section, the termination of Algorithm 2 is garanteed only when the state space A_G is finite and does not contain any cycles. Obviously, in practice, Algorithm 2 is very useful for efficiently exploring the state space of any concurrent system, whether its state space is acyclic or not.

Finally note that Algorithm 2 is different from the algorithms combining persistent sets and sleep sets that appeared in [God96]. Indeed, with a state-less search, different sleep sets associated with the same global state (corresponding to different visits of that state via different paths from s_0) cannot interfere with each other during the search. Moreover, cycles cannot be detected in the context of a state-less search, which makes the use of the provisos discussed in [God96] impossible.

Results of experiments with Algorithm 2 for the dining-philosophers example are presented in Figure 4. Thanks to the use of persistent sets and sleep sets, the run-time explosion of the state-less search is now avoided. Moreover, they yield a significant reduction in the number of transitions that need be explored. Although Algorithm 2 does not store any state in memory, it explores fewer transitions than Algorithm 1!

5 VeriSoft

We have implemented a state-less search using persistent sets and sleep sets in *VeriSoft*, a tool for systematically exploring the state space of systems composed of several concurrent processes executing arbitrary C code. Every process of the concurrent system to be analyzed is mapped to a UNIX process. The execution of the system processes is controlled by an external process, called the *scheduler*. This process observes the visible operations performed by processes inside the system, and can suspend their execution. By resuming the execution of (the next visible operation of) one selected system process in a global state, the scheduler can explore one transition in the state space A_G of the concurrent system. The scheduler also contains an implementation of a search algorithm similar to Algorithm 2. In order to prevent the state-less search from getting lost in cycles of the state space being explored, the depth of the search is limited. When a deadlock or an assertion violation is detected, the search is stopped, and a scenario formed by all the transitions currently stored in *Stack* is exhibited to the user. An interactive graphical simulator/debugger is also available for following the execution of the processes of the system.

In addition to deadlocks and assertion violations, VeriSoft also checks for *divergences* and *livelocks*. A "divergence" occurs when a process does not attempt to execute any visible operation for more than a given (user-specified) amount of time, while a "livelock" occurs when a process has no enabled transition during a sequence of more than a given (user-specified) number of successive global states. Note that these definitions of divergence and livelock differ from the standard definitions for these notions, which correspond to *liveness* properties, i.e., properties that can only be violated by *infinite* sequences of operations or transitions [Lam77, MP92]. In contrast, our notions of divergence and livelock can be violated by *finite* sequences of operations or transitions, and therefore are actually *safety* properties. Indeed, a state-less search cannot detect cycles, and is thus restricted to the verification of safety properties.

At the time of this writing, VeriSoft is being used for analyzing the correctness of several examples of implementations of communication protocols. As an example of application, VeriSoft successfully discovered an error in a 2500-line concurrent C program controlling robots operating in an unpredictable environment. More precisely, this program represents a concurrent system composed of six processes that communicate via shared memory and semaphores. Two of the processes control robots that collect objects randomly dropped on a table by a third robot, represented by a third process. The three other processes are used to simulate the rest of the environment of the robots. Sometimes, a strange behavior of the system can be observed: the two robots that collect objects on the table suddently stop moving.[3] As is often the case with concurrent systems, this phenomenon is extremely hard to reproduce, and seems to occur spontaneously from time to time. After exploring the state space of this system for a few minutes, VeriSoft reported a scenario composed of 29 transitions (as defined in Section 2) that led to a divergence. After replaying this scenario at the C level using the VeriSoft simulator, it was easy to see that the problem was caused by an error in a "while" loop in the C code for one of the processes, and to understand under which circumstances the execution of that process was trapped inside the loop. The divergence in that process would then block the other processes of the system that were waiting for it to proceed.

[3]Actually, a seventh process is used to visualize on the screen the position of all the objects and of the arms of the robots on the table; this process does not influence the behavior of the other system processes.

6 Conclusions and Comparison with Related Work

We have presented a new search technique for efficiently exploring the state space of concurrent systems composed of processes described by programs written in full-fledged programming languages such as C or C++. For finite acyclic state spaces, we showed that our algorithm can be used for detecting deadlocks and assertion violations without incurring the risk of any incompleteness in the verification results. In practice, our algorithm can be used for systematically and efficiently testing the correctness of any concurrent system, whether its state space is acyclic or not. This algorithm is built upon existing state-space pruning techniques known as partial-order methods [God96]. It extends the scope of verification by state-space exploration from modeling languages to programming languages.

Model checking is complementary to other approaches to program analysis. For instance, *static analysis* techniques (e.g., [CC77, MJ81, ASU86]) automatically extract information about the dynamic behavior of a sequential program by examining its text. Variants of these techniques have also been proposed for the analysis of concurrent programs written in concurrent programming languages such as Ada (e.g., [Tay83, LC91, MR93, Cor96]). For specific classes of concurrent programs, these abstraction techniques can produce a "conservative" model of the system that preserves basic information about the communication patterns that can take place in the system. Analyzing such a model using standard model-checking techniques can then prove the absence of certain types of errors in the system. In contrast, our approach is based on the *dynamic* obervation of the "actual" processes of the concurrent system. This makes possible a much closer examination of the behaviors of the system, and the detection of subtle errors that would be missed by the above techniques. Moreover, we do not rely on any specific assumption about the static structure of the programs used to represent the behavior of processes, which can actually be written in any language, or even be unavailable. Interesting future work is to combine the strengths of both the static and dynamic approaches.

Another related and complementary area of research concerns the design of simulators and debuggers for distributed and parallel programs (e.g., [CMN91]). These tools are used to monitor the execution of concurrent processes running in their actual environment. Work in this area discuss techniques for, among others, (1) instrumenting the execution of processes while minimizing the impact of the instrumentation on the timing (scheduling) between the different processes, for (2) storing a minimum amount of information for faithfully replaying ("roll-back") very long scenarios leading to errors, and for (3) obtaining a consistent representation of a state ("snapshot") of a distributed/concurrent system. Note that these problems are avoided with our approach since (1) all the sources of nondeterminism are fully controlled by a scheduler process, (2) the purpose of our approach is to make possible the systematic analysis of short executions of a concurrent system, rather than analyzing very long ones (e.g., containing millions of process transitions), and (3) our analysis is performed by examining only the global states of the concurrent system, which the scheduler process can easily re-create.

Acknowledgments

I wish to thank Thomas Ball, Glenn Bruns, Lalita Jagadeesan, David Weiss, and the anonymous referees for helpful comments on this paper. More information concerning VeriSoft can be found at
`http://www.bell-labs.com/people/god`.

References

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[ASU86] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the 5th Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[BG96] B. Boigelot and P. Godefroid. Model checking in practice: An analysis of the ACCESS.bus protocol using SPIN. In *Proceedings of Formal Methods Europe'96*, volume 1051 of *Lecture Notes in Computer Science*, pages 465–478, Oxford, March 1996. Springer-Verlag.

[Bry92] R.E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages*, January 1977.

[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.

[CGH+93] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, and L. A. Ness. Verification of the Futurebus+ cache coherence protocol. In *Proceedings of the Eleventh International Symposium on Computer Hardware Description Languages and Their Apllications.* North-Holland, 1993.

[CGL92] E.M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, January 1992.

[CMN91] J.-D. Choi, B. P. Miller, and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, pages 491–530, October 1991.

[Cor96] J. C. Corbett. Constructing abstract models of concurrent real-time software. In *Proceedings of ISSTA'96 (International Symposium on Software Testing and Analysis)*, pages 250–260, San Diego, January 1996.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 1(15):36–72, 1993.

[DDHY92] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *1992 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, Cambridge, MA, October 1992. IEEE Computer Society.

[FGM+92] J.C. Fernandez, H. Garavel, L. Mounier, A. Rasse, C. Rodriguez, and J. Sifakis. A toolbox for the verification of LOTOS programs. In *Proc. of the 14th International Conference on Software Engineering ICSE'14*, Melbourne, Australia, May 1992. ACM.

[FHS95] A. R. Flora-Holmquist and M. Staskauskas. Formal validation of virtual finite state machines. In *Proc. Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, pages 122–129, Boca Raton, April 1995.

[GHP95] P. Godefroid, G. J. Holzmann, and D. Pirottin. State-Space Caching Revisited. *Formal Methods in System Design*, 7(3):1–15, November 1995.

[God90] P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. 2nd Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 176–185, Rutgers, June 1990. Springer-Verlag. Extended version in ACM/AMS DIMACS Series, volume 3, pages 321–340, 1991.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, January 1996.

[GP93] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449, Elounda, June 1993. Springer-Verlag.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2):149–164, April 1993.

[HK90] Z. Har'El and R. P. Kurshan. Software for analytical development of communication protocols. *AT&T Technical Journal*, 1990.

[Hol85] G. J. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(12):2413–2434, 1985.

[Hol91] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[JJ91] C. Jard and Th. Jeron. Bounded-memory algorithms for verification on-the-fly. In *Proc. 3rd Workshop on Computer Aided Verification*, volume 575 of *Lecture Notes in Computer Science*, Aalborg, July 1991. Springer-Verlag.

[KP92] S. Katz and D. Peled. Defining conditional independence using collapses. *Theoretical Computer Science*, 101:337–359, 1992.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[LC91] D. L. Long and L. A. Clarke. Data flow analysis of concurrent systems that use the rendezvous model of synchronization. In *Proceedings of ACM Symposium on Testing, Analysis, and verification (TAV4)*, pages 21–35, Vancouver, October 1991.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[Maz86] A. Mazurkiewicz. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency, Advances in Petri Nets 1986, Part II; Proceedings of an Advanced Course*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer-Verlag, 1986.

[McM93] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[MJ81] S.S. Muchnick and N.D. Jones. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer-Verlag, 1992.

[MR93] S. P. Masticola and B. G. Ryder. Non-concurrency amalysis. In *Proceedings of Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel programming*, pages 129–138, San Diego, May 1993.

[Ove81] W. T. Overman. *Verification of Concurrent Systems: Function and Timing.* PhD thesis, University of California Los Angeles, 1981.

[Pel93] D. Peled. All from one, one for all: on model checking using representatives. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423, Elounda, June 1993. Springer-Verlag.

[QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int'l Symp. on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer-Verlag, 1981.

[Rud92] H. Rudin. Protocol development success stories: Part I. In *Proc. 12th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*, Lake Buena Vista, Florida, June 1992. North-Holland.

[Tay83] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, pages 362–376, May 1983.

[Val91] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer-Verlag, 1991.

[VW86] M.Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*, pages 322–331, Cambridge, June 1986.

A Correctness Proofs

Theorem 1 *Consider a concurrent system as defined in Section 2, and let A_G denote its state space. Then, all the deadlocks that are reachable after the initialization of the system are global states, and are therefore in A_G. Moreover, if there exists a state reachable after the initialization of the system where an assertion is violated, then there exists a global state in A_G where the same assertion is violated.*

Proof: (Sketch)
By definition, a deadlock is a state where the execution of the next operation of every process in the system is blocking. Since we assumed that only executions of visible operations may be blocking, all deadlocks are global states.

Let s be a reachable state where an assertion a is violated. Let P_i be the process containing the assertion a. We know that the next operation to be executed by P_i in s is the assertion a, which is a visible operation. For every process P_j other than P_i, let o_j denote the next visible operation that process P_j will eventually execute. Consider the global state s' where, for all processes P_j, $j \neq i$, the next operation to be executed by P_j is the visible operation o_j, and the next operation of process P_i is the assertion a. Clearly, the global state s' is reachable from state s. Moreover, since only invisible operations may have been executed from s to s', assertion a is still violated in s'. (The execution of invisible operations in a process may not change the value of any variable or data structure local to another process.) Finally, since s' is reachable from s, there exists a concurrent execution of the system that reaches the global state s' after the initialization of the system. Any sequence w of process transitions such that the sequence of visible operations in w can be observed during the concurrent execution leading to s' defines a path from s_0 to s' in the global state space A_G of the system. Therefore, s' is in A_G. ■

Let us now turn to the proof of Theorem 2. To establish this result, we use the notion of Mazurkiewicz's traces [Maz86]. *Traces* are defined as equivalence classes of sequences of transitions. Given a set $\mathcal{T}$ and a valid dependency relation $D \subseteq \mathcal{T} \times \mathcal{T}$ as defined in Definition 1, two sequences over $\mathcal{T}$ belong to the same trace with respect to D (are in the same equivalence class) if they can be obtained from each other by successively exchanging adjacent transitions which are independent according to D. For instance, if t_1 and t_2 are two transitions of $\mathcal{T}$ which are independent according to D, the sequences t_1t_2 and t_2t_1 belong to the same trace. A trace is represented by one of its elements enclosed within brackets and, when necessary, subscripted by the alphabet $\mathcal{T}$ and the dependency relation. Thus the trace containing both t_1t_2 and t_2t_1 could be represented by $[t_1t_2]_{(\mathcal{T},D)}$. A trace corresponds to a partial ordering of symbol occurrences and contains all linearizations of this partial order. If two independent symbols occur next to each other in a sequence of a trace, the order of their occurrence is irrelevant since they occur concurrently in the partial order corresponding to that trace.

We will also make use of the two following lemmas from [God96]. These two lemmas state basic properties of persistent sets.

Lemma 4.2 of [God96] *Let s be a state in A_G, and let d be a deadlock reachable from s in A_G by a nonempty sequence w of transitions. For all $w_i \in [w]$, let t_i denote the first transition of w_i. Let Persistent_Set(s) be a nonempty persistent set in s. Then, at least one of the transitions t_i is in Persistent_Set(s).*

Lemma 6.8 of [God96] *Let s be a state in A_G, and let w be a nonempty sequence of transitions from s in A_G. For all $w_i \in [w]$ from s in A_G, let t_i denote the first transition of w_i. Let Persistent_Set(s) be a nonempty persistent set in s. If none of the t_i are in Persistent_Set(s), then all the transitions in Persistent_Set(s) are independent with all the transitions in w.*

To establish the correctness of Algorithm 2, we first prove the following lemma. Assume that all that concerns sleep sets in Algorithm 2 is not implemented (or equivalently that the sleep set associated to every global state reached during the search is empty). We now prove that, under this assumption, if there exists a sequence of transitions in A_G from s_0 to a deadlock or to a state s where an assertion a is violated, then Algorithm 2 without using sleep sets will eventually visit this deadlock or a state where the assertion a is violated, provided that A_G is finite and acyclic.

Lemma 1 *Consider a concurrent system as defined in Section 2, and let A_G denote its state space. Assume A_G is finite and acyclic. Let A_R be the state space explored by Algorithm 2 without using sleep sets. Let s be a state in A_R. Let d be a deadlock reachable from s in A_G by a sequence w of transitions. Then, d is also reachable from s in A_R. Moreover, if s' is a state where an assertion a is violated that is reachable from s in A_G by a sequence w' of transitions, then there exists a state (not necessarily s') reachable from s in A_R where the assertion a is violated.*

Proof:

The proof proceeds by induction on the length of w and w'. For $|w| = 0$ and $|w'| = 0$, the result is immediate. Now, assume the theorem holds for paths (sequences of transitions) of length $n \geq 0$ and let us prove that it holds for paths of length $n + 1$.

Assume a deadlock d can be reached from s by a path w of length $n + 1$ in A_G. For all $w_i \in [w]$, let t_i denote the first transition of w_i. Let Persistent_Set(s) be the nonempty persistent set that is selected in s by Algorithm 2, i.e., the set of transitions that are explored from s in A_R. By Lemma 4.2 of [God96], we know that at least one of the transitions t_i is in Persistent_Set(s). Since t_i is in Persistent_Set(s), it is explored from state s and a state from which a path of length n leads to the deadlock d is reached in A_R. This together with the inductive hypothesis proves the lemma for the deadlock case.

We now consider the case of an assertion violation. Assume that a state s' where an assertion a is violated can be reached from s by a path w' of length $n + 1$ in A_G. Let Persistent_Set(s) be the nonempty persistent set that is selected in s by Algorithm 2, i.e., the set of transitions that are explored from s in A_R. For all $w'_i \in [w']$, let t'_i denote the first transition of w'_i. If at least one of the transitions t'_i is in Persistent_Set(s), it is explored from state s and a state from which a path of length n leads to s' is reached in A_R.

Otherwise, by applying Lemma 6.8 of [God96] to s and w', we know that all the transitions in Persistent_Set(s) are independent with all the transitions in w'. Consequently, for all the states s_j reached after executing one of the transitions in Persistent_Set(s) in A_R, the sequence of transition w' is still executable from s_j in A_G and leads to a state s'_j where the assertion a is violated (this follows from Definition 1). By applying the same reasoning to any state s_j and since all the executions of the system are finite (since its state space is finite and acyclic), one concludes that a transition t'_i is eventually executed from a successor state s_k of s such that all the transitions from s to s_k are independent with all the transitions in w'. After the execution of t'_i from s_k, a state s_l is reached in A_R from which a path of length n in A_G leads to a state where the assertion a is violated. This together with the inductive hypothesis proves the lemma for the case of an assertion violation. ∎

From Lemma 1 it is then immediate to conclude that a state-less search using only persistent sets and started in the initial state of A_G will detect all the deadlocks and assertion violations in A_G. We now show that the use of sleep sets as described in Algorithm 2 preserves this result.

Theorem 2 *Consider a concurrent system as defined in Section 2, and let A_G denote its state space. Assume A_G is finite and acyclic. Then, all the deadlocks in A_G are visited by Algorithm 2. Moreover, if there exists a global state in A_G where an assertion is violated, then there exists a global state visited by Algorithm 2 where* *the same assertion is violated.*

Proof:

Consider a deadlock d or a state s' where an assertion is violated that is reachable from the initial global state s_0. Imagine that we fix the order in which transitions selected in a given state are explored and that we first run Algorithm 2 without sleep sets. Let A_R be the state space explored during this run. Assume that, for every state s in A_R, the transitions explored from s are sorted from left to right following the order in which there are explored: t_1 is to the left of t_2 if t_1 is explored before t_2. Then, we run Algorithm 2 with sleep sets while still exploring transitions in the same order. The important point is that the order used in both runs is the same, the exact order used is irrelevant. By Lemma 1, we know that, if d is a deadlock, d is visited by Algorithm 2 without sleep sets, while if an assertion a is violated in s', a state s'' where the same assertion is violated is visited by Algorithm 2 without sleep sets. We now prove that the leftmost path in A_R leading to d or to a state where the assertion a is violated is still explored in the second run when using Algorithm 2 with sleep sets.

Let $p = s_0 \stackrel{t_0}{\rightarrow} s_1 \stackrel{t_1}{\rightarrow} s_2 \ldots s_{n-1} \stackrel{t_{n-1}}{\rightarrow} s$ be this path. The only reason why it might not be fully explored (i.e., until s is reached) by the algorithm using sleep sets is that some transition t_i of p is not taken because it is in the sleep set associated with s_i. This means that t_i has been added to the sleep set associated with some previous state of the path p and then passed along p until s_i. Let us prove that this is impossible.

Assume that t_i is in the sleep set associated with state s_i, denoted $s_i.Sleep$. Hence, t_i has been added to the sleep set associated with some previous state s_j, $j < i$, of the path p and passed in the sleep set associated with the successor states of s_j along the path p until s_i. Formally, $t_i \notin s_j.Sleep$ when s_j is visited along this path and $t_i \in s_k.Sleep$ for all states s_k, $j < k \leq i$. This implies that t_i has been explored *before* t_j from s_j since a transition is introduced in the sleep set after it has been explored (line 14 of Algorithm 2). Moreover, all transitions that occur between t_j and t_i in p, i.e., all t_k such that $j \leq k < i$, are independent with respect to t_i. Indeed, if this was not the case, t_i would not be in $s_i.Sleep$ since transitions that are dependent with the transition taken are removed from the sleep set (line 11 of Algorithm 2).

Consequently, $t_i t_j \ldots t_{i-1}$ (the sequence $t_j \ldots t_{i-1} t_i$ where t_i has been moved to the first position) is in $[t_j \ldots t_{i-1} t_i]$. Thus, $t_i t_j \ldots t_{i-1}$ and $t_j \ldots t_{i-1} t_i$ are two interleavings of a single trace, and hence lead to the same state: $s_j \stackrel{t_i t_j \ldots t_{i-1}}{\Rightarrow} s_{i+1}$. Since there is a path $s_j \stackrel{t_i t_j \ldots t_{i-1}}{\Rightarrow} s_{i+1}$ from s_j, and since t_i is explored before t_j in s_j, the application of Lemma 1 to the state reached after the execution of t_i from s_j implies that the path p is not the leftmost path in A_R leading to d or to a state where the assertion a is violated. A contradiction. ∎

Finally, it is worth noticing that all the above results also hold when a valid *conditional* dependency relation is used. Moreover, in that case, the above results hold without requiring the valid conditional dependency relation to be *weakly uniform* [God96].