EXPLOITING SYMMETRY WHEN MODEL-CHECKING SOFTWARE

(EXTENDED ABSTRACT)

Patrice Godefroid
Bell Laboratories, Lucent Technologies
263 Shuman Boulevard
Naperville, IL 60566, USA
email: god@bell-labs.com

Abstract We study how to exploit symmetry induced by identical processes or data structures when systematically exploring the state spaces of concurrent software applications such as implementations of communication protocols. Existing model-checking symmetry reduction methods are based on equivalence classes of states, and assume that every system state can easily be encoded by a unique string of bits. When dealing with processes described by software programs written in full-fledged programming languages such as C, C++ or Java, 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 amount of information is typically far too large and complex to be efficiently computed at each step of a state-space exploration.

We develop in this paper a simple theory based on equivalence classes of sequences of transitions for representing symmetries in a system. We then present a state-space exploration algorithm for exploiting symmetries on transitions which does not rely on explicit encodings of system states. This algorithm can be used to dynamically prune the state spaces of implementations of concurrent reactive software systems in a reliable way.

Keywords: software testing, model checking, state-explosion problem, symmetry, partial-order methods.

1 INTRODUCTION

Systematic state-space exploration, also often referred to as "model checking", is an effective method for checking the correctness of concurrent reactive systems. State-space exploration tools for software systems have traditionally been restricted to the exploration of the state space of an abstract description of the system, specified in a modeling language (e.g., [15, 5, 22]). Once a model of a new software application has been thoroughly analyzed, it can also be used as the core of the implementation of the application, as can be done with software development environments for languages such as SDL [16] and VFSM [9].

VeriSoft [11] is a recent tool which extends the scope of systematic state-space exploration to concurrent systems in which processes execute arbitrary code written in general-purpose programming languages such as C or C++. VeriSoft systematically explores the state space of a concurrent software application by controlling and observing the execution of the actual code of all its components. By broadening the scope of systematic state-space exploration from modeling languages to programming languages, VeriSoft eliminates one major obstacle to a wider use of these techniques, namely the need to build a model of the software application to be analyzed. This new tool and approach have been applied successfully for analyzing several software products developed in Lucent Technologies, such as telephone-switching applications and implementations of network protocols (e.g., see [12]). Since VeriSoft can typically generate, execute and evaluate thousands of tests per minute, it can quickly reveal behaviors that are virtually impossible to detect using conventional testing techniques.

A key originality of VeriSoft is that it systematically explores state spaces without storing any explicit encoding of intermediate states in memory. Indeed, when dealing with processes described by software programs written in full-fledged programming languages such as C, C++ or Java, the fundamental assumption used in traditional model-checking that every system state can easily be encoded by a unique string of bits is not valid anymore. Since 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), computing unambiguous encodings of system states and then saving these in memory is typically far too complex and expensive to be performed at each step of a state-space exploration, even for simple software applications.

We investigate in this paper how to exploit symmetry induced by identical processes or data structures for verifying properties of a software application without completely exploring its state space. Existing model-checking symmetry reduction methods [2, 8, 17] define equivalence classes of system states from symmetries in a (model of a) system. During state-space exploration, the successors of a state need not be explored if the successors of an equivalent "symmetric" state have already been explored. Unfortunately, detecting symmetric states requires nontrivial manipulations of state encodings which would not be tractable in the context considered here.

We develop in this work an alternative approach that makes it possible to exploit symmetries in a system without manipulating and comparing encodings of its reachable states. This paper is organized as follows. In the next two sections, we briefly recall the main principles of the framework introduced in [11] and of existing symmetry reduction methods. In Section 4, we describe how to transpose equivalence relations on states to equivalence relations on sequences of transitions for representing symmetries in a system. We then study properties of the transposition. In Section 5, we discuss how to exploit symmetries on transitions in practice. We then present in Section 6 a state-space exploration algorithm that can avoid exploring symmetric parts of a state space without using any explicit encoding of system states. This algorithm uses an original combination of partial-order and symmetry methods. We illustrate our ideas with a simple example in Section 7, and present our conclusions in Section 8.

2 BACKGROUND

We briefly recall in this section the main ideas of the framework introduced in [11]. We consider a concurrent system $\mathcal{S}$ composed of a finite set of *processes* and a finite set of *communication objects*. Each process executes a sequence of *operations* described in a sequential program written in a full-fledged programming language such as C or C++. Such sequential programs are deterministic: every execution of the program on the same input data performs the same sequence of operations. We assume that processes communicate with each other by performing operations on communication objects, such as shared variables, semaphores, and FIFO buffers. We assume that operations on communication objects are executed atomically: namely, no process can execute an operation on a given communication object while another process is currently doing so. 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 currently be completed; for example, waiting for the reception of a message blocks until a message is received. We assume that only visible operations may be blocking.

A concurrent 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. Every process in the system is always expected to eventually attempt execution of a visible operation. (If a process does not attempt to perform a visible operation within a given amount of time, an error can be reported at run time.) This assumption implies that initially, after the creation of all the processes of the system, the system can reach a first and unique global state s_0, called the *initial global state* of the system. A *process transition* is one visible operation followed by a finite sequence of invisible operations performed by a single process and ending just before a visible operation. The *state space* of the concurrent system is then defined as the global states that are reachable from the initial global state s_0, and of the transitions that are possible between these.

It has been proved [11] that *deadlocks* and *assertion violations* of a concurrent system that satisfies the above assumptions can be detected by exploring global states only. Deadlocks are states where the execution of the next operation of every process in the system is blocking. Assertions can be specified by the user with the special operation "VS_assert". This operation can be used 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 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.*

VeriSoft is a tool for systematically exploring the state space of a concurrent system as defined above. Systematic state-space exploration is performed by controlling and observing the execution of all the visible operations of all the concurrent processes of the system. 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 between two global states in the state space of the concurrent system. By reinitializing the system, the scheduler can explore alternative paths in the state space. There are exactly two sources of nondeterminism in the concurrent systems we consider here: concurrency and "VS_toss" operations. VS_toss is a special operation which simulates nondeterminism and is convenient for modeling the environment in which an open system operates and for specifying test drivers. Since the VeriSoft scheduler has complete control over nondeterminism, it can always reproduce any scenario leading to an error found during a state-space search.

Since the states of programs can be very complex (because of pointers, dynamic memory allocation, large data structures of various shapes, recursion, etc.), the VeriSoft scheduler does not attempt to compute any representation for the reachable states of the system being analyzed, and hence performs a systematic state-space exploration without storing any intermediate states in memory. It is shown in [11] that the key to make this approach tractable is to use a new search algorithm built upon existing state-space pruning techniques known as partial-order methods [10]. For finite acyclic state spaces, this search algorithm is guaranteed to terminate and can be used for detecting deadlocks and assertion violations without incurring the risk of any incompleteness in the verification results. In practice, VeriSoft can be used for systematically and efficiently testing the correctness of any concurrent system, whether or not its state space is acyclic. Indeed, it can always guarantee, from a given initial state, complete coverage of the state space up to some depth.

3 SYMMETRY ON STATES

The state space of a concurrent system as defined in the previous section can be represented by a *transition system.* Given a nonempty finite set P of *atomic propositions*, we define a transition system as a tuple $M = (S, L, R, s_0)$ where

- S is a set of global states,
- $L : S \rightarrow 2^P$ is an *interpretation* function that associates with each state the set of atomic propositions that are true in the state,
- $R \subseteq S \times S$ is a set of transitions,
- s_0 is the initial global state.

A transition $t = (s, s')$ is said to be *enabled* in state s. The set of enabled transitions in a state s is denoted by $enabled(s)$. 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 recall the main principles of symmetry-based reduction methods [2, 8, 17]. The basic idea is that symmetries in the system induce an equivalence relation on states of the state space of the system. While performing model-checking, one can discard a state s if one has already explored an equivalent state s'. Precisely, we have the following.

A *permutation* σ on a set S is a bijection of S into itself. Let $Perm(S)$ be the group of all permutations of S [13].

Definition 1 Given a transition system $M = (S, L, R, s_0)$, a subgroup G of $Perm(S)$ is called a *symmetry group* of M if for all permutations σ in G,

- $(s_1, s_2) \in R$ iff $(\sigma(s_1), \sigma(s_2)) \in R$, and
- $L(s) = L(\sigma(s))$.

■

The symmetry group G defines an equivalence relation $\equiv$ on S as follows: $s_1 \equiv s_2$ if $\exists \sigma \in G : s_2 = \sigma(s_1)$. The equivalence class $[s] = \{\sigma(s) | \sigma \in G\}$ of s under $\equiv$ is called the *orbit* of s under G.

Definition 2 Given a transition system $M = (S, L, R, s_0)$ and a symmetry group G of M, a *quotient transition system* for M modulo G is a transition system $M_{[s]} = (S', L', R', s'_0)$ where

- $S' = \{[s] | s \in S\}$,
- $L' : S' \to 2^P$ is such that $\forall s \in S : L'([s]) = L(s)$,
- $R' = \{([s], [s']) | (s, s') \in R\}$,
- $s'_0 = [s_0]$.

■

Since $s \equiv s'$ implies $L(s) = L(s')$, atomic propositions do not distinguish equivalent states. It is then easy to prove that s_0 and $[s_0]$ are bisimilar.

Definition 3 Let $M_1 = (S_1, L_1, R_1, s_0^1)$ and $M_2 = (S_2, L_2, R_2, s_0^2)$ be transitions systems. A binary relation $\mathcal{B} \subseteq S_1 \times S_2$ is a *bisimulation relation* if $(s_1, s_2) \in \mathcal{B}$ implies:

- $L_1(s_1) = L_2(s_2)$,
- if $(s_1, s'_1) \in R_1$, then there is some $s'_2 \in S_2$ such that $(s_2, s'_2) \in R_2$ and $(s'_1, s'_2) \in \mathcal{B}$, and
- if $(s_2, s'_2) \in R_2$, then there is some $s'_1 \in S_1$ such that $(s_1, s'_1) \in R_1$ and $(s'_1, s'_2) \in \mathcal{B}$.

Two states s_1 and s_2 are *bisimilar*, denoted $s_1 \sim s_2$, if they are related by some bisimulation relation. ■

Theorem 1 *Let $M = (S, L, R, s_0)$ be a transition system and $M_{[s]} = (S', L', R', s'_0)$ be a quotient transition system for M modulo a symmetry group G of M. Then, we have $s_0 \sim s'_0$.*

It is well-known [14] that bisimilar states cannot be distinguished by formulas of propositional modal fixpoint logic [24], also known as the propositional μ-calculus [19]. Therefore, the quotient transition system $M_{[s]}$ can be used instead of the full transition system M for model-checking formulas of this logic or of any of its fragments, such as linear-time temporal logic (LTL) [20] and computation-tree logic (CTL) [3].

In practice, the quotient transition system of a system is usually generated on-the-fly using a *canonicalization function* ζ. This function maps each state s into a unique representative $\zeta(s)$ of the equivalence class $[s]$. Whenever a state s is visited during state-space exploration, the value $\zeta(s)$ is saved in memory, usually in a hash-table [17]. This type of search can also be done "symbolically", for instance using Binary Decision Diagrams [1] for representing sets of states [2]. When a state s' equivalent to s is visited later during the search, the search does not explore successors of s'. Various schemes have been proposed for efficiently implementing canonicalization functions [17]. The complexity of this problem is discussed in [4].

4 SYMMETRY ON TRANSITIONS

Storing states in memory in one form (hash-table) or another (BDDs) and using canonicalization functions assumes that every system state can easily be encoded by a unique string of bits. When dealing with processes described by arbitrary programs written in full-fledged programming languages, this assumption is clearly not valid anymore.

Of course, nothing prevents us from systematically searching the state space of a concurrent system without storing any intermediate states in memory, as is done in VeriSoft. This is called a *state-less search* in [11]. In the context of a state-less search, a state is identified only by the sequence of transitions that were executed from the initial state s_0 in order to reach that state. In other words, the sequence of transitions that leads to a state from s_0 can be viewed as a "representation" of that state. Obviously, this representation is not unique since many different paths from s_0 may lead to a same state.

We now discuss how symmetry can be exploited for pruning state spaces in conjunction with a state-less search. We start by transposing the equivalence relation on states defined in the previous section to an equivalence relation on transitions.

Definition 4 Let $t = (s, s')$ denote a transition of a transition system M, and let σ denote a permutation of a symmetry group G of M. We write $\sigma(t)$ to denote the transition $(\sigma(s), \sigma(s'))$. We then define the relation $\equiv$ on transitions as follows: $t \equiv t'$ if $\exists \sigma \in G : t' = \sigma(t)$. ■

Lemma 1 *The relation $\equiv$ on transitions is an equivalence relation.*

We can also extend the definition of $\equiv$ on transitions to sequences of transitions as follows.

Definition 5 Let $w = t_1 t_2 \ldots t_n$ denote a nonempty sequence of transitions of a transition system M, and let σ denote a permutation of a symmetry group G of M. We write $\sigma(w)$ to denote the sequence of transitions $\sigma(t_1)\sigma(t_2)\ldots\sigma(t_n)$. We then define the relation $\equiv$ on nonempty sequences of transitions as follows: $w \equiv w'$ if $\exists \sigma \in G : w' = \sigma(w)$. ■

Lemma 2 *The relation $\equiv$ on sequences of transitions is an equivalence relation.*

As before, we denote by $[t]$ the equivalence class $[t] = \{\sigma(t) | \sigma \in G\}$ of t under $\equiv$, while $[w]$ denotes the equivalence class $[w] = \{\sigma(w) | \sigma \in G\}$ of w under $\equiv$.

We define a quotient transition system based on an equivalence relation on sequences of transitions as follows.

Definition 6 Given a transition system $M = (S, L, R, s_0)$ and a symmetry group G of M, a *quotient transition system* for M modulo G defined with an equivalence relation $\equiv$ on sequences of transitions is a transition system $M_{[w]} = (S', L', R', s'_0)$ where

- $S' = \{[w] | s_0 \stackrel{w}{\Rightarrow} s \text{ in } M\}$,
- $L' : S' \rightarrow 2^P$ is such that $L'([w]) = L(s)$ with $s_0 \stackrel{w}{\Rightarrow} s$ in M,
- $R' = \{([w], [wt]) | s_0 \stackrel{w}{\Rightarrow} s \text{ in } M \text{ and } t = (s, s') \in R\}$,
- $s'_0 = \epsilon$ (the empty word).

■

We now show that the initial states of M and $M_{[w]}$ are bisimilar.

Theorem 2 *Let $M = (S, L, R, s_0)$ be a transition system and $M_{[w]} = (S', L', R', s'_0)$ be a quotient transition system for M modulo a symmetry group G of M as defined in Definition 6. Then, $s_0 \sim s'_0$.*

Proof: Proofs of lemmas and theorems are omitted in this extended abstract. ■

Since the initial states of M and $M_{[w]}$ are bisimilar, $M_{[w]}$ can be used for model-checking temporal properties such as those discussed in the previous section.

From the previous theorem and Theorem 1, we immediately obtain by transitivity that the initial states of $M_{[w]}$ and $M_{[s]}$ are also bisimilar. We now compare the sizes of both types of quotient transition systems.

It is easy to prove that equivalent sequences of transitions from the initial state s_0 always lead to equivalent states.

Lemma 3 *Let $M = (S, L, R, s_0)$ be a transition system and let σ be a permutation of a symmetry group G of M. For any sequences w and w' of transitions such that $s_0 \stackrel{w}{\Rightarrow} s$ and $s_0 \stackrel{w'}{\Rightarrow} s'$, $w' = \sigma(w)$ implies $s' = \sigma(s)$.*

Since each equivalence class $[w]$ of sequences of transitions from s_0 can be mapped to the equivalence class $[s]$ of states such that $s_0 \stackrel{w}{\Rightarrow} s$, it is easy to see from Definitions 6 and 2 that $M_{[w]}$ will always contain at least as many states and transitions as $M_{[s]}$. Hence $M_{[w]}$ cannot be smaller than $M_{[s]}$.

In contrast with Lemma 3, it is possible to prove that equivalent states s and $\sigma(s)$ can be reached from s_0 by equivalent sequences of transitions only if $s_0 = \sigma(s_0)$, i.e., s_0 is symmetric with respect to permutation σ in G.

Lemma 4 *Let $M = (S, L, R, s_0)$ be a transition system and let G be a symmetry group of M. If $s_0 = \sigma(s_0)$ with $\sigma \in G$, then for any states s and s' such that $s' = \sigma(s)$, $s_0 \stackrel{w}{\Rightarrow} s$ implies $s_0 \stackrel{\sigma(w)}{\Rightarrow} s'$.*

Putting it all together, we now state the exact condition required for $M_{[w]}$ and $M_{[s]}$ to be identical. ($\Rightarrow$ denotes logical implication.)

Theorem 3 *Let $M = (S, L, R, s_0)$ be a transition system. Let $M_{[w]}$ and $M_{[s]}$ be quotient transition systems for M modulo a symmetry group G of M as defined in Definition 6 and Definition 2 respectively. Then $M_{[w]}$ and $M_{[s]}$ are identical if and only if the two following conditions are satisfied in M:*

1. $\forall s \in S : (s_0 \stackrel{w}{\Rightarrow} s \wedge s_0 \stackrel{w'}{\Rightarrow} s) \Rightarrow (\exists \sigma \in G : w' = \sigma(w))$.
2. $\forall \sigma \in G : (s_0 \stackrel{w}{\Rightarrow} s \wedge s_0 \stackrel{w'}{\Rightarrow} \sigma(s)) \Rightarrow (\exists \sigma' \in G : s_0 = \sigma'(s_0) \wedge \sigma(s) = \sigma'(s))$.

$M_{[w]}$ and $M_{[s]}$ are identical when there is a one-to-one correspondence between equivalence classes on states and equivalence classes on sequences of transitions. This is the case for instance when the state space is isomorphic to a tree and $|[s_0]| = 1$ (i.e., the initial state is symmetric with respect to every permutation in G).

In summary, exploring exactly one sequence w of transitions per equivalence class $[w]$ is sufficient to visit at least once every equivalence class of states $[s]$ and transitions $[t]$ in a transition system. If exactly one sequence w per equivalence class $[w]$ is explored, redundant visits of a same equivalence class of states or transitions are due only to the exploration of multiple paths that are not equivalent with respect to $\equiv$. In practice, such redundant visits are usually unavoidable with a state-less search since multiple unequivalent paths may very well lead to the same state or to equivalent states.

5 EXPLOITING SYMMETRY ON TRANSITIONS

In order to exploit the ideas presented in the previous section, we need a practical way to determine when transitions (elements of R) are symmetric without computing and comparing explicit encodings of the states they originate from or lead to. We propose to use a labeling function for representing actions performed during the execution of transitions, in conjunction with permutation functions on actions for modeling symmetries in the system.

Precisely, we first extend the definition of transition systems to allow labels on transitions.

Definition 7 A *labeled transition system* (LTS) $M = (S, L, R, s_0, \Sigma, \ell)$ is a transition system (S, L, R, s_0) extended with a set Σ of *actions*, and a labeling function $\ell : R \rightarrow \Sigma$. ■

The action of a transition t, denoted $\ell(t)$, identifies the visible operation executed during the transition. This identifier includes the type of visible operation (system call) executed, values of parameters if any, and the identity of the process executing the operation. By construction, we guarantee that, for any action a in Σ and any state s in S, there is at most one transition labeled with a from s. Therefore, any transition from a given state can be uniquely identified by its label. Note that the following results also hold for transitions (pairs of states) labeled with more than one label, as long as these labels satisfy the previous assumption.

Symmetries in the system can then be modeled by permutation functions π_σ on actions (elements of Σ). Specifically, a function π_σ on actions is associated with each permutation in G as follows.

Definition 8 Let $M = (S, L, R, s_0, \Sigma, \ell)$ be a LTS and let σ be a permutation of a symmetry group G of M. A partial function $\pi_\sigma : \Sigma \rightarrow \Sigma$ on actions is a *valid permutation function on actions* for permutation σ in G iff the following property holds for all actions a in Σ:

- if $\pi_\sigma(a)$ is defined, then $\forall t = (s, s') \in R : \ell(t) = a \;\Rightarrow\; \ell(\sigma(t)) = \pi_\sigma(a)$.

■

By definition, $\pi_\sigma(a)$ is unique when it is defined.

In practice, the user can declare symmetries in a system by providing values of permutation functions π_σ on actions. Standard classes of permutations can be pre-packaged in order to facilitate this step. For instance, various classes of process symmetries can easily be specified using simple manipulation functions on process indices (such as swapping, circular rotation, etc.) which can be applied uniformly to any action name, independently of the type of operation executed. An example of function π_σ on actions is given in Section 7.

Of course, declaring symmetries to the verification system has to be done with care since the validity of the verification results relies on the validity of these assumptions. Conservative approximations that are sufficient for identifying symmetric transitions but may miss other symmetries can be used.

The following rule exploits symmetries on actions in order to prune transitions during state-space exploration.

Definition 9 (Pruning Rule) In any state s reached during state-space exploration, for any pair of transitions t and t' enabled in s such that

$$\exists \sigma \in G : \ell(t') = \pi_\sigma(\ell(t)) \wedge s = \sigma(s),$$

explore only one of these two transitions from state s. ■

The correctness of this rule is based on the following theorem.

Theorem 4 *Let s be a state of a LTS $M = (S, L, \Sigma, R, s_0)$ and let G be a symmetry group on M. If t and t' are transitions in enabled(s) such that $\exists \sigma \in G : \ell(t') = \pi_\sigma(\ell(t)) \wedge s = \sigma(s)$, then $t' = \sigma(t)$.*

From Theorem 2 of the previous section, we already know that exploring (sequences of) transitions that are equivalent with respect to $\equiv$ is unnecessary in order to verify properties as those considered in Section 3. When the condition stated in Definition 9 is satisfied, we have $t \equiv t'$ by Theorem 4, and it is thus sufficient to explore only one of these transitions.

In order to implement the rule above, we need a practical way to check whether a reachable state s satisfies the condition $s = \sigma(s)$ without using any explicit encoding of s. We now discuss how this can be done assuming we know that $s_0 = \sigma(s_0)$.

Let s be a state reachable from s_0 by a nonempty sequence $w = t_1 t_2 \ldots t_n$ of transitions. Assuming $s_0 = \sigma(s_0)$, $s = \sigma(s)$ can be proved by showing that $s_0 \stackrel{\sigma(w)}{\Rightarrow} s$. If for all $1 \leq i \leq n$, $\ell(t_i) = \pi_\sigma(\ell(t_i))$, then $t_i = \sigma(t_i)$ for all t_i in w, and $s = \sigma(s)$. Otherwise, let $t_i = (s_{i-1}, s_i)$ denote the first transition along w such that $\ell(t_i) \neq \pi_\sigma(\ell(t_i))$, and let $t'_i = \sigma(t_i)$. Since t_i is the first such transition in w and since $s_0 = \sigma(s_0)$, we know that $s_{i-1} = \sigma(s_{i-1})$. Therefore, t'_i is enabled in s_{i-1} and labeled by $\pi_\sigma(\ell(t_i))$, but was not explored because of the pruning rule defined above. In other words, the exploration of $\sigma(w)$ stopped in s_{i-1}. Checking that $\sigma(w)$ leads from s_0 to s may thus seem problematic.

Fortunately, there is a way to check whether $\sigma(w)$ leads to s without using any explicit encoding for s or without even executing $\sigma(w)$ in the first place: if w and $\sigma(w)$ are different interleavings of a same partial ordering of transitions, then they are guaranteed to lead to a same state, and hence $s = \sigma(s)$. In the next section, we use partial-order methods to solve this problem and complete our algorithm.

6 COMBINING SYMMETRY AND PARTIAL-ORDER REDUCTIONS

Partial-order methods [10] denote another family of algorithms for tackling the state explosion problem which limits the efficiency and applicability of verification by state-space exploration. The basic idea behind partial-order methods is that many paths in the state space of a concurrent system correspond simply to different execution orders of the same process transitions (defined in Section 2). If process transitions are "independent", for instance because they are executed by noninteracting processes, then changing their execution order will not modify their combined effect. This notion of independency between process transitions and its complementary notion, the notion of dependency, can be formalized by the following definition (adapted from [18]).

Definition 10 Let $\mathcal{T}$ be the set of process transitions of the system 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 state space 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 process 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 process transitions).

■

This definition characterizes the properties of possible "valid" dependency relations for the process transitions of a given system. In practice, it is possible to give easily checkable syntactic conditions that are sufficient for process transitions to be independent. In a concurrent system as defined in Section 2, dependency can arise between transitions of different processes that access the same communication objects. We refer the reader to [10] for a detailed presentation of this topic.

Following the work of Mazurkiewicz [21], one can use the notion of independent process transitions to define an equivalence relation on sequences of process transitions: two sequences of process transitions are equivalent if they can be obtained from each other by successively permuting adjacent independent process transitions. Thus, given a valid dependency relation, sequences of process transitions can be grouped into equivalence classes which Mazurkiewicz calls traces. By construction, all the sequences of process transitions of a Mazurkiewicz trace executed from a given state all lead to the same final state (e.g., see Theorem 3.10 in [10]).

Thanks to this property and assuming we know $s_0 = \sigma(s_0)$, we can check whether $s_0 \stackrel{\sigma(w)}{\Rightarrow} s$ by checking whether w and $\sigma(w)$ (computed by applying π_σ successively to the labels of the transitions in w) can be obtained from each other by successively permuting adjacent independent process transitions. In the worst-case, this can be done in time $O(|P||w|)$ where $|P|$ is the number of processes in the system and $|w|$ is the length of w. Since the purpose of the approach developed in this work is to make possible the systematic analysis of all executions of a concurrent system up to some depth, w is usually short (e.g.,

```
1   Initialize: Stack is empty;
2   Search() {
3     DFS(∅);
4     }
5   DFS(set: Sleep) {
6      T = Persistent_Set()\Sleep;
7      T = Sym(T, Stack);
8      while T ≠ ∅ do {
9          take t out of T;
10         push (t) onto Stack;
11         Execute(t);
12         DFS({t' ∈ Sleep | t' and t are independent});
13         pop t from Stack;
14         Undo(t);
15         Sleep = Sleep ∪ {t};
16         };
17     }
```

Figure 1: State-less search using symmetry and partial-order reductions

less than a few hundred process transitions), and the computation is fast. If the result of this check shows that w and $\sigma(w)$ belong to the same Mazurkiewicz trace, we can safely conclude that $s = \sigma(s)$. Otherwise, we have to assume by default that $s \neq \sigma(s)$. This prevents the pruning of any transition in s using permutation σ.

Putting it all together, we can now present in Figure 1 a complete algorithm for performing a state-less search using symmetry and partial-order reductions. This algorithm is an extension of the algorithm of [11]. The only difference is the addition of line 7. This algorithm performs a selective depth-first search (DFS) in the state space of a concurrent system and uses two partial-order reduction techniques: *persistent sets* and *sleep sets*. A detailed presentation of these algorithmic techniques can be found in [10, 11]. The data structure *Stack* contains the sequence of process transitions that leads from the initial global state s_0 to the current global state being explored. A set *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 process 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. The rules for computing the sleep set of a new state are given in lines 12 and 15.

In lines 6 and 7, a new set of process transitions is selected to be explored from s in two steps. Persistent_Set() returns a persistent set in the current global state s that is nonempty if there exist process transitions enabled in s.

In line 7, symmetric process transitions are eliminated using the new function Sym described in details below. In line 11, a process 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 process transitions of *Sleep* that are independent with t are passed into the sleep set associated to that new global state (line 12). Once the search from that new state (and hence the corresponding call to DFS) is completed, the exploration of the other process transitions selected to be explored from s may proceed. The concurrent system is then brought back to the global state s in line 14. (This can be done by reinitializing the system and reexecuting the sequence of process transitions in *Stack*, for instance.) Next, process transition t, i.e., the last process transition explored from s, is added to *Sleep* in line 15. Note that this algorithm does not compute, store or manipulate any explicit state encoding.

The originality of the algorithm of Figure 1 is thus the use of the function Sym in line 7. The function Sym checks whether any process transition of its first argument T can be pruned using the rule of Definition 9. It takes as arguments a subset T of the set of enabled process transitions in s and the sequence of process transitions formed by the current depth-first search *Stack*. When the function Sym finds a permutation σ such that $\ell(t') = \pi_\sigma(\ell(t))$ with t and t' in T, it then checks whether $s = \sigma(s)$ by examining the current path *Stack* of process transitions being explored and its symmetric image $\sigma(Stack)$ following the procedure described earlier in this section. The function Sym finally returns a subset (not necessarily proper) of its argument T which determines the process transitions to be explored from the current state s.

When both symmetry and partial-order reduction methods are applied, the properties preserved by their combination is the intersection of the properties preserved by each type of reduction alone. Since the algorithm of [11] preserves deadlocks and assertion violations, let us now discuss how the use of symmetry reductions affects this specific class of properties. By definition of a symmetry group, if a global state s is a deadlock, so are all the states s' in $[s]$. Thus, exploring at least one state per equivalence class $[s]$ using symmetry reductions is sufficient to detect a deadlock in the system. In contrast, the detection of an assertion violation using symmetry methods is guaranteed only for assertions that are violated in all the states of an equivalence class $[s]$. Let us call such assertions "symmetry-preserving assertions". For instance, any assertion that does not distinguish a process from any other is symmetry-preserving under any process symmetry.

Formally, the correctness of the algorithm of Figure 1 is established by the following theorem.

Theorem 5 *Consider a concurrent system as defined in Section 2, and assume its state space is finite and acyclic. Then, if there exist deadlocks in the state space of the system, the algorithm of Figure 1 will visit at least one of them. Moreover, if there exists a global state in the state space of the system where a symmetry-preserving assertion is violated, then there exists a global state visited by the algorithm of Figure 1 where the same assertion is violated.*

```
/* 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);
}
```

Figure 2: A simple example of concurrent C program

7 EXAMPLE

In this section we illustrate our ideas with a simple example. Consider the concurrent C program shown in Figure 2. 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

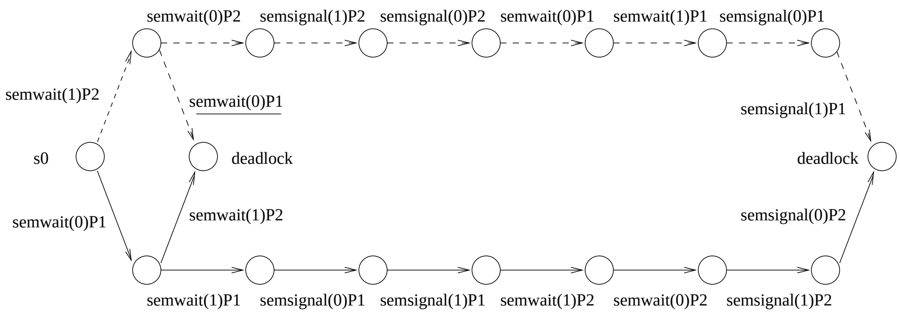

Figure 3: State space of the two-dining-philosophers system

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 operation *exit* is a visible operation whose execution is always blocking.

The state space of this system is shown in Figure 3, where global transitions are labeled with the visible operation of the corresponding process transition. Since all the processes are deterministic, nondeterminism (i.e., branching) in A_G is caused only by concurrency.

The dining-philosophers system contains symmetry: swapping the states of the two processes and semaphores yields an equivalent state. We can model this symmetry by a permutation function π_σ on actions as defined in Definition 8[1]:

- $\pi_\sigma(semwait(x)_{P1}) = semwait((x+1)\%2)_{P2}$,
- $\pi_\sigma(semwait(x)_{P2}) = semwait((x+1)\%2)_{P1}$,
- $\pi_\sigma(semsignal(x)_{P1}) = semsignal((x+1)\%2)_{P2}$,
- $\pi_\sigma(semsignal(x)_{P2}) = semsignal((x+1)\%2)_{P1}$,
- $\pi_\sigma(exit(0)_{P1}) = exit(0)_{P2}$, and
- $\pi_\sigma(exit(0)_{P2}) = exit(0)_{P1}$.

When using this permutation function with the state-space exploration algorithm of Figure 1, the dotted transitions of Figure 3 are not explored thanks to symmetry reductions. For this example, half of the transitions need not be explored with the new algorithm, which is the maximum reduction one can obtain with this type of process symmetry and two processes. Note that, for this example, using partial-order reduction techniques (persistent sets and sleep sets) alone would avoid exploring only one transition in the state space (the transition whose label is underlined is pruned thanks to sleep sets).

Table 1 compares the number of transitions explored by a state-less search, a state-less search using partial-order reductions (SL+PO) as described in [11],

[1]The modulus binary operator % returns the remainder from the division of its first argument by its second argument.

N	State Space	State Less	SL+PO	SL+PO+SYM
2	(18)	18	17	9
3	(123)	1,680	75	37
4	(708)	386,816	275	133
5	(3,765)	?	959	461

Table 1: Number or transitions explored

and a state-less search using partial-order reductions and symmetry reductions (SL+PO+SYM) as described in Figure 1. Results are given for the previous program with various numbers N of processes. The numbers between parentheses in the column entitled "State Space" (column 2) give the total number of transitions in the state space. The run time of the three algorithms considered is proportional to the number of explored transitions.

One clearly sees from the numbers of transitions explored by a simple state-less search (column 3) that the run-time cost of not storing visited states in memory is very high. However, this cost can be dramatically reduced by using partial-order reduction techniques (column 4), as extensively discussed in [11]. Moreover, the use of symmetry reductions yields additional substantial reductions (column 5). For this example, the number of explored transitions is reduced roughly by a factor of 2 when exploiting symmetry.

Note that the maximum theoretical symmetry reduction with this type of process symmetry is by a factor of N, where N is the number of processes in the system, since the size of each equivalence class of states and transitions is at most N. In contrast, partial-order reductions reduce the number of explored transitions of a state-less search by a factor much greater than N for $N > 2$ in this example. Therefore, a state-less search using symmetry reductions alone (i.e., without partial-order reductions) could not yield such a huge reduction.

8 CONCLUSIONS

We have developed a simple framework for exploiting symmetry in a software application in order to safely avoid exploring parts of its state space when searching for deadlocks and assertion violations. Our framework is based on equivalence classes of sequences of transitions instead of the traditional equivalence classes of states. We then presented a state-space exploration algorithm for exploiting symmetries on transitions which does not rely on explicit encodings of system states. This algorithm combines both symmetry and partial-order reduction methods in an original way. It can be used to systematically and efficiently explore the state spaces of implementations of concurrent reactive software systems.

Note that it has been known for some time that symmetry and partial-order reduction methods are essentially orthogonal, and hence compatible and complementary. The combination of state-based symmetry reductions with partial-order reductions was already suggested and studied in [23, 6, 7].

ACKNOWLEGMENTS

I wish to thank Ramin Hojati, Norris Ip, Lalita Jagadeesan, Denis Leroy and Prasad Sistla for interesting discussions on combining partial-order and symmetry methods. I am also thankful to Glenn Bruns and the anonymous referees for helpful comments on this paper.

References

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

[2] E. Clarke, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 450–462, Elounda, June 1993. Springer-Verlag.

[3] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching-Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[4] E. M. Clarke, E. A. Emerson, S. Jha, and A. P. Sistla. Symmetry Reductions in Model Checking. In *Proc. 10th Conference on Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 147–158, Vancouver, June/July 1998. Springer-Verlag.

[5] 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.

[6] S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Application and Experimental Evaluation of State Space Reduction Methods for Deadlock Analysis in Ada. *ACM Transactions on Software Engineering and Methodology,* 3(4):340–380, October 1994.

[7] E. A. Emerson, S. Jha, and D. Peled. Combining Partial Order and Symmetry Reductions. In *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 19–34, Twente, April 1997. Springer-Verlag.

[8] E. A. Emerson and A. P. Sistla. Symmetry and model checking. In *Proc. 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 463–478, Elounda, June 1993. Springer-Verlag.

[9] 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.

[10] P. 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.

[11] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186, Paris, January 1997.

[12] P. Godefroid, R. S. Hanmer, and L. J. Jagadeesan. Model Checking Without a Model: An Analysis of the Heart-Beat Monitor of a Telephone Switch using VeriSoft. In *Proceedings of ACM SIGSOFT ISSTA'98 (International Symposium on Software Testing and Analysis)*, pages 124–133, Clearwater Beach, March 1998.

[13] J. A. Green. *Sets and Groups – A First Course in Algebra*. Routledge and Kegan Paul, 1965.

[14] M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

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

[16] G. J. Holzmann and J. Patti. Validating sdl specifications: An experiment. In *Proc. 9th IFIP WG 6.1 International Symposium on Protocol Specification, Testing, and Verification*. North-Holland, 1989.

[17] C. N. Ip and D. L. Dill. Better verification through symmetry. In D. Agnew, L. Claesen, and R. Camposano, editors, *Proceedings of the 1993 Conference on Computer Hardware Description Languages and their Applications*, April 1993.

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

[19] D. Kozen. Results on the Propositional Mu-Calculus. *Theoretical Computer Science*, 27:333–354, 1983.

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

[21] 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.

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

[23] A. Valmari. Stubborn sets for of colored petri nets. In *Proceedings of the 12th International Conference on Apllications and Theory of Petri Nets*, pages 102–121, Gjern, 1991.

[24] M.Y. Vardi. Why is modal logic so robustly decidable? In *Proceedings of DIMACS Workshop on Descriptive Complexity and Finite Models.* AMS, 1997.