A Symbolic Partial Order Reduction Algorithm for Rule Based Transition Systems

Ritwik Bhattacharya, Steven German,
Ganesh Gopalakrishnan

UUCS-03-028

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

December 1, 2003

Abstract

Partial order reductions are a class of methods that attempt to reduce the state space that must be explored to verify systems by explicit state enumeration. Partial order reduction algorithms have been successfully incorporated into tools such as Spin and VFSM-valid. However, current partial order algorithms assume that the concurrency model is based on processes. Rule based formalisms, such as Unity and Murphi, are another important and widely used class of modeling techniques. Many important types of systems, such as distributed shared memory (DSM) protocols, are best modeled as a set of global transitions. Rule-based systems require a new approach to implementing partial order reduction, since traditional heuristics are not applicable. Also, the traditional methods of computing the dependence relation give approximations that cause many potential reductions to be missed. We propose a novel algorithm based on using a SAT solver to compute the dependence relation, and a new heuristic for computing ample sets for rule based formalisms.

A Symbolic Partial Order Reduction Algorithm for Rule Based Transition Systems[⋆]

Ritwik Bhattacharya[1], Steven German[2], and Ganesh Gopalakrishnan[1]

[1] School of Computing, University of Utah
{ritwik, ganesh}@cs.utah.edu
[2] IBM T.J. Watson Research Center
german@watson.ibm.com

Abstract. *Partial order reductions* are a class of methods that attempt to reduce the state space that must be explored to verify systems by explicit state enumeration. Partial order reduction algorithms have been successfully incorporated into tools such as Spin and VFSM-valid. However, current partial order algorithms assume that the concurrency model is based on processes. Rule based formalisms, such as Unity and Murphi, are another important and widely used class of modeling techniques. Many important types of systems, such as distributed shared memory (DSM) protocols, are best modeled as a set of global transitions. Rule-based systems require a new approach to implementing partial order reduction, since traditional heuristics are not applicable. Also, the traditional methods of computing the dependence relation give approximations that cause many potential reductions to be missed. We propose a novel algorithm based on using a SAT solver to compute the dependence relation, and a new heuristic for computing ample sets for rule based formalisms.

1 Introduction

Partial order reduction is a technique that curbs state explosion in model checking based on explicit state enumeration. Partial order reductions[23,7] are based on the observation that much of the state explosion in verifying concurrent systems stems from the interleaving of independent actions. Partial order methods formalize the notion of redundant interleavings and attempt to generate a subset of all the interleavings that is sufficient to check a given property. The efficacy of partial order reductions has been established both theoretically and in practical experiments[8]. Partial order reduction algorithms have been incorporated into tools such as Spin[11,13] and VFSM-valid[9].

[⋆] This work was supported in part by NSF Awards CCR-0081406 and CCR-0219805, and SRC Contract 1031.001

Most previous work on partial order reduction deals with concurrency models based on processes. In the process-based approach, concurrency is modeled as the interleaving of a set of sequential processes with local variables; the processes communicate via shared global variables and message passing.

In many cases, verification models of complex systems do not easily fit into the communicating process model. For example, with distributed shared memory protocols, it is difficult to decide what constitues a process – should it be a hardware node, in which case many of its "local" data structures have to be made globally available, or should it be a message, in which case too its buffers need to be made global. The essential difficulty lies in the fact that a large percentage of the state variables are truly global.

Rule-based formalisms provide a natural method for describing many complex real systems. Indeed, languages such as Unity [2] and Murphi [4] are widely used to model concurrency for such systems.

In this paper, we address two important problems in applying partial order reduction to complex rule-based systems. First, as mentioned above, most previous approaches have considered process models of concurrency.

A second shortcoming of traditional partial order reduction algorithms is their reliance on syntactic methods for computing the dependence relation. Syntactic methods use occurences of variables to determine dependence instead of solving for the possible values of variables. In order to be sound, syntax-based dependence computation algorithms must conservatively overapproximate the dependence relation, which can result in missing possible reductions.

The new results of this paper address both of these issues. First, we enable reduction of rule based systems by presenting a new *ample set* calculation heuristic that is applicable to rule based formalisms. Second, we develop a new symbolic dependence calculation algorithm which uses a SAT solver. We have implemented both of these algorithms in a new version of the Murphi verification tool called POeM – partial order enabled Murphi – with encouraging results.

The SAT based algorithm, which is also applicable to other, non-rule-based systems, provides a more accurate characterization of the independence relation than the traditional syntactic approach, which can often lead to greater reductions.

As an illustration of the advantage of our approach, consider the simple Murphi system shown in 1. In the Murphi language [5], a model consists of a set of global state variables and a set of conditional transitions. Transitions have the form

`rule` "`name`" *guard* ==> *body*.

A transition system is executed as follows: at each step in the execution, select a single transition whose guard is true in the current state and

```
CONST
   NumNodes: 2;
   NumBufs : 1;
VAR
   i: 0..2;
   a: 0..3;
   b: array [0..2] of boolean;

rule "1"
True
 ==>
a := (a + 1) % 4;
end;

rule "2"
True
 ==>
a := (a + 2) % 4;
end;

rule "3"
(i >= 0)
 ==>
b[i+1] := true;
end;

rule "4"
(i <= 0)
 ==>
b[i+2] := false;
end;
```

Fig. 1. Example Murphi system

then execute the statements in the body. The state reached by executing the body is taken as the next state of the system.

A syntax-based approach would classify rules 1 and 2 as dependent on each other, since they both update the same variable `a`. In the figure, the binary operator `n % 4` gives the value of its first argument mod 4. Symbolic evaluation quickly discovers that the values of `a` after applying both rules 1 and 2 in either order are identical, and hence, the SAT based approach will rightly classify the rules as independent. Similarly, rules 3 and 4 in the figure, which both access the same array variable, would be classified as dependent by the syntax-based approach, whereas, the SAT based approach will find that they are independent, since they access different indices of the array `a`.

Before proceeding to the details of our techniques, it is worth commenting on the continuing practical importantance of explicit state enumeration in model checking. Even with the recent impressive advances in BDD-based and SAT-based model-checking, a large class of systems cannot yet be model-checked effectively using such Boolean representations. For example, in case of BDD-based model-checking applied to the formal verification of cache coherence protocols, the sheer number of variables, the extensive use of high level data structures such as arrays, and the 'all to all' dependencies between the variables in these protocols have resulted in BDDs exceeding current capacities [14].

Thus, there is still a great need for improving the capabilities of explicit state enumeration model checkers. Explicit state enumeration model checkers are also the mainstay of tools in *software verification*, another important emerging area. A recent survey of industrial formal methods [10] shows the continued widespread use of model-checkers such as Murphi and TLA in the industry.

1.1 Related Work

There has been extensive research on partial order reduction methods (see [8] for a good survey). Few previous works address reduction for formalisms without processes. TLC, the explicit state model checker for TLA+ [17], is reported to have a partial order reduction algorithm [18], but we are not aware of the details of the algorithm used. Partial order reduction algorithms have also been proposed for symbolic state exploration methods [1]. The algorithm there is based on a modified breadth first search, since symbolic state exploration is essentially breadth first. The *in-stack check* of the traditional partial order algorithm is replaced by a check against the set of visited states. An alternative to the traditional runtime ample set computation algorithm is discussed in [16]. The idea is to statically compute all of the partial order reduction data, and generate a compiled model that is already reduced, which then allows the use of any model checking back-end for the actual verification. However, this suffers from the same problem as other static algorithms, in that only a limited amount of information is available at compile time.

The technique we use to transform the Murphi language (and its sequential programming constructs) into propositional logic is similar to the well known method in which sequential programs are verified by constructing formulas called *verification conditions*. The Stanford Pascal Verifier is a representative implementation of this technique. Several recently developed tools convert C-language programs into SAT problems, for instance, for purposes of comparing a C program and a Verilog program [15]. We are not aware of previous use of symbolic simulation and SAT to check independence relations.

The rest of the paper is organized as follows. Section 2 introduces relevant terminology, and gives an overview of explicit state enumeration based model checking and the partial order reduction technique. In Section 3, we describe the new algorithm, and sketch a proof of its correctness. Experiments and results are discussed in Section 4, with conclusions and future directions in Section 5.

2 Partial Order Reduction

2.1 Notations and Definitions

A labeled finite state system $\mathcal{F}$ is a 5-tuple $\langle S, T, I, P, L\rangle$ where S is a finite set of *states*, T is a finite set of deterministic *transitions*, such that every $t \in T$ is a partial function $t : S \mapsto S$, $I \subseteq S$ is the set of initial states, P is a set of atomic propositions, and $L : S \mapsto 2^P$ labels each state with a set of propositions that are true in the state.

A *labelled path* of a finite state system is an infinite sequence starting with a state and then alternating transitions and states,

$$s_0, t_0, s_1, t_1, s_2, t_2, \ldots,$$

where $\forall i \geq 0 : t_i(s_i) = s_{i+1}$. A labelled path is called a labelled *run* if it starts with a state in I. For any labelled path p of a system, we define the predicate $\mathbf{before}(p, t_1, t_2)$ to be true when t_1 occurs before the earliest occurrence of t_2 in p, or t_2 does not occur in p. Let the set of all labelled paths of a finite state system be $\mathcal{P}$. The *restriction* of $\mathcal{P}$ with respect to a state s, written $\mathcal{P}_{|s}$, is the set of all labelled paths in $\mathcal{P}$ starting from the state s. A transition t is said to be *enabled* at a state s if $\exists s' \in S : t(s) = s'$. We define the predicate $\mathbf{en}(s,t)$ that is true exactly when t is enabled at s. We also define the predicate $\mathbf{enabled}(s) = \{t \in T \mid \mathbf{en}(s,t)\}$. Two transitions t_1 and t_2 are *independent* iff the following conditions hold:

- *Enabledness*: $\forall s \in S : \mathbf{en}(s,t_1) \wedge \mathbf{en}(s,t_2) \Rightarrow \mathbf{en}(t_1(s),t_2) \wedge \mathbf{en}(t_2(s),t_1)$
- *Commutativity*: $\forall s \in S : \mathbf{en}(s,t_1) \wedge \mathbf{en}(s,t_2) \Rightarrow t_1(t_2(s)) = t_2(t_1(s))$

We define the predicate $\mathbf{ind}(t_1, t_2)$, that is true exactly when t_1 and t_2 are independent, and $\mathbf{dep}(t_1,t_2) = \neg\mathbf{ind}(t_1,t_2)$. A *property* π of a system is a formula in next-time free LTL (LTL$_{-X}$), such that the set of propositions in the logic is P. For any property π, we define $\mathbf{props}(\pi) \in 2^P$ as the set of propositions occurring in π. A transition t is *invisible* with respect to a property π, written as $\mathbf{inv}_\pi(t)$, iff:

$$\forall s_1, s_2 \in S : t(s_1) = s_2 \Rightarrow L(s_1) \cap \mathbf{props}(\pi) = L(s_2) \cap \mathbf{props}(\pi)$$

2.2 Explicit State Enumeration Based Model Checking

Explicit state enumeration based model checking algorithms explore the entire state space of a given transition system, by traversing the state graph of the system, while checking that the specified properties hold in the system. In this paper, we only consider algorithms that explore the state space by a *depth-first search* (dfs). Figure 2 shows a typical routine to perform a dfs exploration of a state space. As can be seen from the figure, the dfs routine maintains a stack, which holds the set of states on the current path being explored. We define a dynamic function **onstack**$(s) : S \mapsto 2^S$ that returns the stack at any given point in a run of the dfs algorithm on a system, when the state currently being explored is s.

The main drawback of explicit state enumeration based methods is that most real systems have very large state spaces that cannot be fully explored without exhausting the resources available, the most important resource constraint being the amount of physical memory available on the computer system that the model checking is being carried out. Therefore, various ideas have been explored to reduce the state space that needs to be searched, while still preserving the correctness of the algorithm. Partial order reduction is one such technique.

```
proc DFS(s) {
  push(s,stack);
  while(!empty(stack)) {
    current_state := top(stack);
    stack := pop(stack);
    trans_set := enabled(current_state);
   while(!empty(trans_set)) {
      t := select(trans_set);
      trans_set := trans_set - t;
      next_state := t(s);
      if (!in_hash_table(next_state)) {
        store(next_state,hash_table);
        push(next_state,stack);
      }
    }
  }
}
```

Fig. 2. DFS routine for state space exploration

2.3 Partial Order Reductions

The basic idea behind partial order reductions is that, in the interleaving model of execution of asynchronous communications systems, there is an arbitrary ordering imposed on the executions of concurrently enabled transitions. Without partial order reductions, all possible interleavings of concurrent transitions are explored while enumerating the state space of the system. But exploring *all* interleavings of *invisible* and *independent* transitions is usually unnecessary to prove the required properties of the system. Partial order reductions characterize a sufficient subset of the interleavings with respect to the property being proved.

Under certain assumptions about the system, invisibility and independence of transitions allow us to avoid generating all interleavings, and thus reduce the number of states generated while exploring the state space. A number of partial order algorithms have been devised that are based on this concept, which differ somewhat in the details. In this paper, we focus on the *ample-set construction method*, first proposed in [20]. This method proceeds by performing a modified depth-first search where, at each state, a subset of all the enabled transitions is chosen, called the *ample set*. That is, $\mathbf{ample}(s) \subseteq \mathbf{enabled}(s)$. Transitions from the ample set are then the only ones that are taken out of that state to explore the state space. This leads to a subset of the entire state space being explored. Naturally, it is important to ensure that this subset preserves the property of interest, *i.e.*, if the original graph contains a violating execution, there is a corresponding violating execution in the reduced graph. Thus, for each path in the full graph, there must be a *representative* path in the reduced graph. The following conditions, adapted from [3], guarantee the existence of such representative paths:

- **C0 :** $\forall s \in S : \mathbf{ample}(s) = \phi \Leftrightarrow \mathbf{enabled}(s) = \phi$.
- **C1:** $\forall s \in S : \forall t_1, t_2 \in T : t_1 \in \mathbf{ample}(s) \wedge t_2 \notin \mathbf{ample}(s) \wedge \mathbf{dep}(t_1, t_2) \Rightarrow \forall p \in \mathcal{P}_{|s} : \exists t_3 \in \mathbf{ample}(s) : \mathbf{before}(p, t_3, t_2)$
- **C2 :** $\forall s \in S : \mathbf{ample}(s) \neq \mathbf{enabled}(s) \Rightarrow \forall t \in \mathbf{ample}(s) : \mathbf{inv}_\pi(t)$.
- **C3**[1] **:** $\forall s \in S : \mathbf{ample}(s) \neq \mathbf{enabled}(s) \Rightarrow \exists t \in \mathbf{ample}(s) : t(s) \notin \mathbf{onstack}(s)$

Condition **C0** states that the ample set at a state is empty iff the set of enabled transition is empty. Condition **C2** says that if the transitions at a state are not fully expanded, all of the transitions in the ample set must be *invisible*. Condition **C3** requires that there be at least one transition in an ample set that leads to a state not on the current dfs stack, which ensures that atleast one transition in the ample set does not create a cycle. This version of **C3** is sufficient for the verification of safety properties [12]. **C1**, also known as the *faithful decomposition* condition [21], is the most complex, and states that for all paths starting from a state s in the full state graph, it must never be the case that a transition dependent on some transition in the ample set at s is taken before a transition from the ample set. The intuitive reason for this condition is

[1] For a proof of the necessity and sufficiency of this form of the condition see [12]

that if the only paths not in the sub-graph all contain a sequence of independent transitions leading up to a transition in the ample set, then those paths can be shown equivalent to paths in the sub-graph. For a detailed motivation of the conditions, see [3, chapter 10].

Thus, an implementation of the ample-set based partial order reduction algorithm is guaranteed to be correct if it satisfies the above conditions.

In the following section, we describe our algorithms for computing the independence relation and ample sets, and sketch a proof of how the algorithms satisfy the conditions above.

3 Implementing Partial Order Reductions for Murphi

Murphi was developed at Stanford[4], and is used extensively in both academia and industry for the verification of real world protocols. It provides a fairly complex rule-based system description language, including arrays, functions and procedures, and a subset of LTL (*linear-time temporal logic*) is used to specify properties of interest. Murphi includes a number of features that provide state space reduction, including symmetry-based reductions, state caching, and hash compaction.

Let the finite state system described by a Murphi model M be $\mathcal{F}(M) = \langle S_M, T_M, I_M, P_M, L_M \rangle$. A *state* of $\mathcal{F}(M)$ is an evaluation of all of the *state variables* of the system. Each rule or transition t is a pair $\langle g_t, a_t \rangle$, where g_t, the *guard*, is some predicate over the state variables, and a_t, the *action*, is a partial function $S_M \mapsto S_M$ defined by a program that assigns new values to the state variables. Let $a_t(s)$ be the state reached when rule t is executed in state s, provided $g_t(s)$ is true. The execution of a rule corresponds to the taking of a transition in $\mathcal{F}(M)$. A rule t is *enabled* at a state s, if its guard evaluates to true at that state, *i.e.*, $\mathbf{en}(s,t) \Leftrightarrow g_t(s)$. Therefore, the *enabledness* and *commutativity* conditions of section 2.1 can now be written as:

$$\forall s \in S_M : \forall t_1, t_2 \in T_M : g_{t_1} \wedge g_{t_2} \Rightarrow g_{t_2}(a_{t_1}(s)) \wedge g_{t_1}(a_{t_2}(s)) \tag{3.1}$$

$$\forall s \in S_M : \forall t_1, t_2 \in T_M : g_{t_1} \wedge g_{t_2} \Rightarrow a_{t_2}(a_{t_1}(s)) = a_{t_1}(a_{t_2}(s)) \tag{3.2}$$

Computing the independence relation thus requires evaluating the above conditions for each pair of rules in the system.

3.1 Computing the Independence Relation

In traditional partial order reduction algorithms, the above conditions are tested by examining the state variables occurring in transitions. This approach leads to an overapproximation of the dependence relation. For complex rule-based systems, the dependence relation has to be determined more accurately to yield useful model reduction. We do this by encoding the relations 3.1 and 3.2 as boolean formulas, and using a SAT solver to check validity. As a first step, we take the programs defining the guards and next-state functions, and symbolically evaluate the executable statements to produce formulas over finite data types. We are guaranteed to be able to perform this evaluation since Murphi only allows the description of finite state systems. We then decide the validity of the generated formulas, by encoding the variables in a straightforward way into bit vectors, and using a SAT solver.

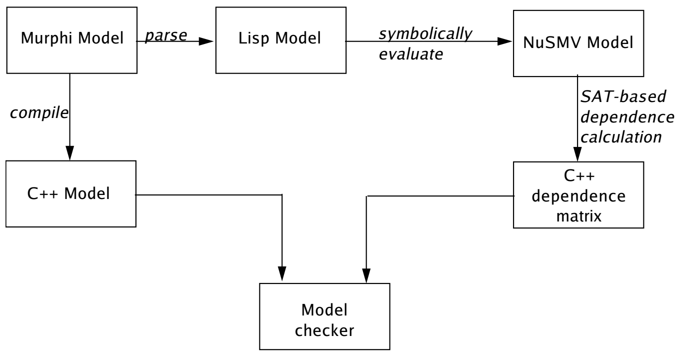

Fig. 3. Murphi verification flow with partial order reduction

Implementation Figure 3 shows how the above idea has been implemented in the Murphi system. In the original Murphi flow, system descriptions are compiled to a C++ program which incorporates the checker. To implement the dependence calculation, we also translate the Murphi description into Lisp s-expressions. Our symbolic simulator tool then generates symbolic expressions corresponding to the *commutativity* and *enabledness* conditions of each pair of transitions. The simulator emits NuSMV code representing these expressions, with one file per condition per pair of rules. The specification for each NuSMV program is either a commutativity or an enabledness condition. NuSMV is convenient since it provides high-level data types such as enumerated types and integer subranges, which are automatically encoded internally into boolean variables. NuSMV is then used to invoke the SAT solver zchaff, which checks the specification. The results of the zchaff runs are automatically analyzed to produce a fragment of C++ code representing a *dependency matrix*, which records, for each pair of rules, whether they are dependent or not. This code is included in the C++ program generated by the Murphi compiler. The Murphi checker has been suitably modified to use this dependency matrix at runtime to generate the ample set at each state, if partial order reductions are enabled. How the ample set is constructed is described in the next section.

3.2 Ample Set Computation

Recall that in the traditional partial order algorithm, the ample set is constructed by selecting a process at each state, all of whose enabled transitions are independent of transitions from all other processes. If such a process is not available, the state is fully expanded, and all transitions from all processes are explored. Since Murphi does not provide the notion of processes, the above method is inapplicable.

```
proc ample(s) {
  ample := { pick_new_invisible(enabled(s)) };
  if (empty(ample))
   return enabled(s);
  while (exists_dependent(enabled(s),ample)) {
   ample := ample + all_dependent(enabled(s),ample);
  }
  if ((ample = enabled(s)) or exists_visible(ample))
   return enabled(s);
  for (t_d in disabled(s)) {
   if (dependent(t_d, ample))
    return enabled(s);
  }
  for (t_a in ample) {
   if (!in_stack(t_a(s)))
    return ample;
  }
  return enabled(s);
}
```

Fig. 4. Ample set construction algorithm for Murphi

Our algorithm for computing the ample set is shown in Figure 4. Intuitively, it picks an enabled, invisible transition (called the *seed transition*) at each state, and tries to form an ample set using this transition. To do this, it first adds all enabled transitions that are dependent on transitions already in the ample set, and computes the transitive closure of the dependence relation with respect to transitions in the ample set. Then, it checks to see whether there are *disabled* transitions that are dependent on a transition in the ample set. If this is true, then the ample set may be invalid, because there may exist a path in the full graph that takes this dependent transition before any transition from the ample set is taken,

which violated condition C1. Therefore, we return the set of all enabled transitions. However, if this is not true, then we may have a successful ample set, as long as at least one of the transitions in the ample set leads to a state on on the current stack. If this is the case, we return this ample set. Otherwise, we return the set of all enabled transitions.
Note that the invisibility condition can be encoded as a propositional formula whose validity can be determined by using a SAT solver, in a similar fashion as the independence checks of section 3.1.

Proof of Correctness of Ample Set Construction

In this section, we sketch a proof of correctness of the algorithm presented above. We are required to show that the algorithm satisfies the conditions **C0-C3** of section 2.3. We examine each condition in turn.

- **C0 :** $\forall s \in S : \mathbf{ample}(s) = \phi \Leftrightarrow \mathbf{enabled}(s) = \phi$.
  - $\leftarrow$ First, assume that the set of enabled transitions at some state is empty. The check on line `3` will succeed, and the algorithm will return as the ample set the set of all enabled transitions, which in this case is the empty set. Thus, when the set of enabled transitions is empty, so is the ample set.
  - $\rightarrow$ Now, assume that the ample set returned at some state is empty. If the algorithm returned from one of lines `4`,`9`,`12` or `18`, then it actually returned the set of enabled transitions, so the set of enabled transitions must also be empty. If not, it must have returned from line `16`. Observe that, between lines `5` and `16`, the algorithm does not remove anything from the ample set it is attempting to build. Therefore, if it returned an empty set from line `16`, `ample` must have been empty at line `5`. However, if the algorithm reached line `5`, it must have failed the emptiness check at line `3`, which means that `ample` could not have been empty at line `5`. This contradicts our assumption that the algorithm returned the empty set from line `16`. Thus, if the algorithm returned an empty set, it must be the case that the set of enabled transitions at that state was also empty.
- **C1:** $\forall s \in S : \forall t_1, t_2 \in T : t_1 \in \mathbf{ample}(s) \wedge t_2 \notin \mathbf{ample}(s) \wedge \mathbf{dep}(t_1, t_2) \Rightarrow \forall p \in \mathcal{P}_{|s} : \exists t_3 \in \mathbf{ample}(s) : \mathbf{before}(p, t_3, t_2)$

  Assume that the antecedent holds. That is:

  $$\exists s \in S : \exists t_1, t_2 \in T : t_1 \in \mathbf{ample}(s) \wedge t_2 \notin \mathbf{ample}(s) \wedge \mathbf{dep}(t_1, t_2) \tag{3.3}$$

  First, note that if $\mathbf{enabled}(s) = \mathbf{ample}(s)$, then **C1** is trivially satisfied, since the consequent is guaranteed to be true. Therefore, we assume that $\mathbf{ample}(s) \neq \mathbf{enabled}(s)$. In that case, our algorithm must have returned from line `16`. This implies that the check in line `11` failed for every iteration of the loop in lines `10`–`13`. This means that, for every `t_d` in `disabled(s)` (line `10`), it must have been the case that `dependent(t_d,ample)` (line `11`) was false. That is:

  $$\forall t_1, t_2 \in T : t_1 \in \mathbf{ample}(s) \wedge t_2 \notin \mathbf{enabled}(s) \rightarrow \neg(\mathbf{dep}(t_1, t_2)) \tag{3.4}$$

From the loop in lines 5–7, it is clear that, after the loop terminates, there is no enabled transition not in the ample set, that is dependent on a transition in the ample set. In other words:

$$\forall t_1, t_2 \in T : t_1 \in \mathbf{ample}(s) \wedge \mathbf{dep}(t_1, t_2) \rightarrow \neg(t_2 \in \mathbf{enabled}(s)) \tag{3.5}$$

Formulas 3.4 and 3.5 together immediately lead to a contradiction with formula 3.3. Thus, our algorithm guarantees that the antecedent of **C1** never holds, which implies that the implication **C1** is always true.

– **C2 :** $\forall s \in S : \mathbf{ample}(s) \neq \mathbf{enabled}(s) \Rightarrow \forall t \in \mathbf{ample}(s) : \mathbf{inv}_{\pi}(t)$.

  Assume, on the contrary, that for some state s, $\mathbf{ample}(s) \neq \mathbf{enabled}(s) \wedge \exists t \in \mathbf{ample}(s) : \neg\mathbf{inv}_{\pi}(t)$. Once again, we can assume that the algorithm returned from line `16`. Note that the last line at which anything is added to `ample` is line `6`. Therefore, the value of `ample` is unchanged between lines `8` and lines `16`, if line `16` is indeed reached. By our assumption, therefore, at line `8`, `ample` must have contained a transition t, such that $\neg\mathbf{inv}_{\pi}(t)$. But, in that case, the check on line `9` would have succeeded, and we would have returned the set of all enabled transitions, which contradicts our assumption that we returned at line `16`. Therefore, it must be the case that our assumptions are untenable, which implies that our algorithm does indeed satisfy **C2**.

– **C3 :** $\forall s \in S : \mathbf{ample}(s) \neq \mathbf{enabled}(s) \Rightarrow \exists t \in \mathbf{ample}(s) : t \notin \mathbf{onstack}(s)$

  Again, assume that, on the contrary, for some state s,$\mathbf{ample}(s) \neq \mathbf{enabled}(s) \wedge \forall t \in \mathbf{ample}(s) : t(s) \in \mathbf{onstack}(s)$. Like for **C2** above, the algorithm must have returned from line `16`. Therefore, the check at line `15` must have succeeded, which means that $\exists t \in \mathbf{ample}(s) : \neg\, t \in \mathbf{onstack}(s)$. This contradicts our assumption that $\forall t \in \mathbf{ample}(s) : t \in \mathbf{onstack}(s)$. Therefore, it must be the case that our algorithm satisfies **C3**.

Thus, we have demonstrated that our algorithm satisfies all of the conditions **C0**-**C3**, and therefore, that it correctly computes ample sets at each state.

4 Experiments and Results

We have run our partial order algorithm on a few examples[2] of varying sizes, and the results are shown in Table 1. The most significant examples, both in terms of the model size and the reductions obtained, are the instances of Steven German's cache coherence protocol. This is a simple, directory based protocol with a single address, and a variable number

[2] Note to reviewers: we will have more examples by the time the final version of the paper is due

Example	**Number of states**	
	without po	**with symbolic po**
German cache protocol, 3 nodes	28593	8413
German cache protocol, 4 nodes	566649	90636
German cache protocol, 5 nodes	-	990272
Dekker's algorithm	100	90
More examples to be added		

Table 1. Performance of partial order reduction algorithm

of client nodes. The examples were run with invariant checking turned off to do a full state space exploration. For the 5 node instance of the coherence protocol, with partial order reduction turned off, the verifier does not finish, even with 200MB of RAM allocated for the hash table, whereas with reductions enabled, the verifier completes exploration with just 100MB of RAM. We have also implemented a syntactic partial order algorithm for Murphi, which gave no reductions on the above examples.

5 Conclusions and Future Directions

In this paper, we have presented a new partial order algorithm based on a symbolic computation of the independence relation. Combined with a new heuristic to compute ample sets, this provides a means of applying partial order reductions to rule based formalisms, such as Murphi. Our experiments confirm that partial order reductions can greatly reduce the state space that needs to be explored in explicit-state enumeration based algorithms, even for non process-based models of concurrency.

There are a number of improvements that can be made to the current algorithm, which may result in greater reductions. For example, the current algorithm picks the *seed transition* in some pre-determined, but non-intelligent order. Since a commonly used heuristic in partial order reductions is to try and minimize the size of the ample set at each state, it makes sense to try and pick a seed transition that has the fewest dependent transitions. This information can easily be computed from the outputs of the SAT solver, and incorporated into the ample set computation algorithm. Another natural extension to the algorithm would be to parallelize it. There have been other efforts in this direction [19], but these have been in the process context.

Also, a lot of Murphi's advantages as a modeling language are due to its support for functions and procedures, and rich data types such as multi-dimensional arrays and records, which our implementation (ie, the symbolic evaluator) does not yet handle, although we are working on adding support for these features.

Finally, although we currently use SAT as the decision procedure in the symbolic part of our algorithm, it might be more efficient to use a higher level decision procedure for quantifier free formulas with equality, finite arithmetic and arrays. Recently, several theorem provers have been developed that include these domains [22,6] . It would be interesting to try these decision procedures in our algorithm. Another promising idea is that when checking independence of two transitions, it should often be possible to abstract parts of the transitions rather than expanding all of the details. We could use the logic of uninterpreted functions to express such abstractions, and use decision procedures for this logic to decide the formulas.

References

1. Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351, 1997.
2. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.
3. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, December 1999.
4. David Dill. The stanford murphi verifier. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, New Jersey, July 1996. Springer-Verlag. Tool demo.
5. David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *International Conference on Computer Design*, pages 522–525, 1992.
6. C. Flanagan, R. Joshi, X. Ou, and J.B. Saxe. Theorem Proving Using Lazy Proof Explication. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *Computer Aided Verification*, volume 2725 of *Lecture Notes in Computer Science*, pages 355–367. Springer, 2003.
7. Patrice Godefroid. Using partial orders to improve automatic verification methods. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 176–185, New Brunswick, NJ, USA, June 1990. Springer-Verlag.
8. Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag, New York, NY, USA, 1996.
9. Patrice Godefroid, Doron Peled, and Mark G. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. In *International Symposium on Software Testing and Analysis*, pages 261–269, 1996.

10. Ganesh Gopalakrishnan and Warren A. Hunt Jr., editors. *Formal Methods in Systems Design*, volume 22(2). Kluwer, March 2003. Special Issue on Industrial Practice of Formal Hardware Verification: A Sampling.
11. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.
12. G.J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proc. 12th Int. Conf on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, Fl., June 1992.
13. G.J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. Formal Description Techniques, FORTE94*, pages 197–211, Berne, Switzerland, October 1994. Chapman & Hall.
14. Alan John Hu. Techniques for efficient formal verification using binary decision diagrams. Technical Report CS-TR-95-1561, 1995.
15. Daniel Kroening, Edmund Clarke, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.
16. R. Kurshan, V.Levin, M.Minea, D.Peled, and H. Yenigün. Static partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 345–357, Lisbon, Portugal, January 1998. Springer-Verlag.
17. L. Lamport. Specifying concurrent systems with tla, 1999.
18. Stephan Merz. Personal communication.
19. Robert Palmer. Parallelized PV with partial order reductions. Personal communication.
20. Doron Peled. All from one, one for all: On model checking using representatives. In *Computer Aided Verification*, pages 409–423, Elounda, Greece, June 1993.
21. Doron Peled. Partial order reduction: Linear and branching temporal logics and process algebras. In *Partial Order Methods in Verification; DIMACS Workshop*, volume 29, pages 233–258. American Mathematical Society, July 1996. Series in Discrete Mathematics and Theoretical Computer Science.
22. Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A Cooperating Validity Checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002. Copenhagen, Denmark.
23. Antti Valmari. A stubborn attack on state explosion. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 156–165, New Brunswick, NJ, USA, June 1990. Springer-Verlag.

A Murphi Code for German's Cache Protocol [3]

```
CONST
     num_clients : 3;
TYPE
     message : enum{empty, req_shared, req_exclusive, invalidate,
      invalidate_ack, grant_shared, grant_exclusive};

     cache_state : enum{invalid, shared, exclusive};

     client: 1 .. num_clients;
VAR
     channel1: array[client] of message;

     channel2_4: array[client] of message;

     channel3: array[client] of message;

     home_sharer_list: array[client] of boolean;

     home_invalidate_list: array[client] of boolean;

     home_exclusive_granted: boolean;

     home_current_command: message;

     home_current_client: client;

     cache: array[client] of cache_state;

RULESET cl: client do
 RULE "client requests shared access"
      cache[cl] = invalid & channel1[cl] = empty ==>
        begin channel1[cl] := req_shared end;

 RULE "client requests exclusive access"
      (cache[cl] = invalid | cache[cl] = shared ) & channel1[cl] = empty ==>
        begin channel1[cl] := req_exclusive end;

 RULE "home picks new request"
      home_current_command = empty & channel1[cl] != empty ==>
        begin home_current_command := channel1[cl];
              channel1[cl] := empty;
              home_current_client := cl;
              for i: client do
               home_invalidate_list[i] := home_sharer_list[i]
```

[3] Included for reference by program committee - not part of paper

```
            endfor;
      end;

 RULE "home sends invalidate message"
      (home_current_command = req_shared & home_exclusive_granted
       | home_current_command = req_exclusive)
      & home_invalidate_list[cl] & channel2_4[cl] = empty ==>
       begin
        channel2_4[cl] := invalidate;
        home_invalidate_list[cl] := false;
       end;

 RULE "home receives invalidate acknowledgement"
      home_current_command != empty & channel3[cl] = invalidate_ack ==>
      begin
       home_sharer_list[cl] := false;
       home_exclusive_granted := false;
       channel3[cl] := empty;
      end;

 RULE "sharer invalidates cache"
      channel2_4[cl] = invalidate & channel3[cl] = empty ==>
      begin
       channel2_4[cl] := empty;
       channel3[cl] := invalidate_ack;
       cache[cl] := invalid;
      end;

 RULE "client receives shared grant"
      channel2_4[cl] = grant_shared ==>
      begin
       cache[cl] := shared;
       channel2_4[cl] := empty;
      end;

 RULE "client receives exclusive grant"
      channel2_4[cl] = grant_exclusive ==>
      begin
       cache[cl] := exclusive;
       channel2_4[cl] := empty;
      end;

END;

 RULE "home sends reply to client -- shared"
      home_current_command = req_shared
      & !home_exclusive_granted & channel2_4[home_current_client] = empty ==>
      begin
       home_sharer_list[home_current_client] := true;
       home_current_command := empty;
       channel2_4[home_current_client] := grant_shared;
      end;

 RULE "home sends reply to client -- exclusive"
      home_current_command = req_exclusive
      & forall i: client do home_sharer_list[i] = false endforall
      & channel2_4[home_current_client] = empty ==>
      begin
       home_sharer_list[home_current_client] := true;
       home_current_command := empty;
       home_exclusive_granted := true;
       channel2_4[home_current_client] := grant_exclusive;
      end;

STARTSTATE
begin
 for i: client do
   channel1[i] := empty;
   channel2_4[i] := empty;
   channel3[i] := empty;
   cache[i] := invalid;
   home_sharer_list[i] := false;
   home_invalidate_list[i] := false;
  endfor;
home_current_command := empty;
home_current_client := 1;
home_exclusive_granted := false;
end;
```