State space reduction using Predicate Filters

Sujatha Kashyap and Vijay K. Garg [⋆]

Parallel and Distributed Systems Lab.
ECE Department
University of Texas at Austin
Austin, TX 78712, USA
{`kashyap, garg`}`@ece.utexas.edu`

Abstract. In a previous paper, we presented a technique called finite trace covers, in which a program is represented by a set of partial orders. We also presented classes of reachability properties that could be checked in time that is polynomial in the size of the trace cover. A deficiency of this approach was that it did not reduce the complexity of checking properties that did not belong to an efficient class.
In this paper, we present a technique called **predicate filtering**, which can be used as a state space reduction technique to reduce the complexity of checking general reachability properties. To verify a property ϕ that does not belong to one of our efficient property classes, the user can first specify a *weaker* property ψ, which belongs to an efficient class. The program is **filtered** through the property ψ. The result is a set of partial orders, which can be represented as programs, and contain all the states that satisfy ψ, while eliminating most of the states that do not. Traditional model checking techniques can then be used to verify ϕ on the reduced program.
In experiments, we verified a leader election protocol by constructing only one-third as many states as constructed by SPIN using partial order reduction.

1 Introduction

The modelling of concurrency by partial orders has been advocated by many researchers as a means of controlling state space explosion in the verification of concurrent asynchronous systems. The partial order nature of concurrency has been used to guide the construction of a *reduced* (explicit) state transition graph [1–3]. This approach has been dubbed "partial order reduction", and is used with widespread success in the popular model checker SPIN.

The effectiveness of a state space reduction technique depends on how much *less* memory it uses, compared to generating the full state space graph. For p.o. reduction techniques, such as those proposed by Valmari [2], Godefroid [3], Peled [1], and others, the amount of reduction achieved depends on two factors: (1) the

[⋆] supported in part by the NSF Grants CNS-0509024 and an Engineering Foundation Fellowship amount of concurrency in the program, and (2) the proportion of *invisible* events in the program. Given a program and a formula being checked, an event is said to be *invisible* if executing it has no effect on the truth value of *any* proposition in the formula. As observed in [4], the effectiveness of p.o. reduction techniques diminishes rapidly with an increase in the number of visible events.

In [5], we proposed a partial order representation of the state space, called the *finite trace cover*, which is a finite set of (Mazurkiewicz) traces[6], each of finite length, which encodes the reachable state space of the program. Each trace is represented by a partial order on the set of its constituent events. The set of all order ideals of this partial order exactly corresponds to the set of reachable states of the trace. Every finite state program has a finite trace cover.

The trace cover representation avoids state space explosion due to concurrency by avoiding *explicit* construction of the states. The amount of reduction achieved depends **only** on the amount of concurrency in the program. As our experimental results in [5] show, p.o. reduction (as implemented by SPIN) used **10 times** as much memory as our finite trace cover approach in the verification of a distributed mutual exclusion protocol.

Unfortunately, CTL model checking is NP-complete in the size of the finite trace cover [7, 8]. In [5], we presented some *tractable* classes of properties for which model checking can be performed in time that is polynomial in the size of the trace cover. But what of properties that do not belong to any of these tractable classes? This paper addresses that question.

In this paper, we present a powerful technique called *predicate filtering*. In this technique, we first take a program and convert it into its finite trace cover representation. Then, we *filter* each trace in the cover with respect to a (user-specified) predicate ϕ. The *filtrate* is, in a sense to be defined later, the "smallest" program that contains all the states of the trace that satisfy ϕ, while eliminating states that do not satisfy ϕ[1]. The filtrate of a trace can be computed in polynomial time if and only if the specified predicate can be *detected* in polynomial time on the trace. Predicate filtering was first introduced in the context of distributed computations in [9].

For program verification, a user can specify a predicate belonging to a tractable class, that is *weaker* than the actual property to be verified. We can filter the program through this weaker, efficient predicate, and then verify the actual property on the smaller filtrate program. In our experiments, we could verify a leader election protocol using predicate filtering, by constructing only one-third as many states as constructed by SPIN using p.o. reduction.

This paper is organized as follows. Section 2 presents necessary background material and notations. Section 3 explains the relation between traces and lattices, laying the theoretical groundwork for predicate filtering. Section 4 defines finite trace covers, and presents tractable classes of predicates. Section 5 introduces the theory behind predicate filters, and presents an algorithm for computing the filtrate directly from the poset representation of a trace. Section 6

[1] As we will see in Section 5, depending on the predicate ϕ, the filtering process may not eliminate **all** the states that do not satisfy ϕ.

discusses how predicate filters can be used for state space reduction, and presents a case study, with experimental data.

2 Preliminaries

A program $\mathcal{P}$ is a triple (S, T, s_0), where S is a *finite* set of states, T is a finite set of operations, and $s_0 \in S$ is the initial state[2]. The set of *all* operations that can be executed from a state s is denoted by $enabled(s)$. An operation $t \in enabled(s)$ transforms the state s into a *unique* state $s' \in S$. Each occurrence (execution) of an operation is called an *event*. We denote the unique state s' reached upon executing the event α from the state s by $s' := \alpha(s)$.

An interleaving sequence of a program $P = (S, T, s_0)$ is any finite or infinite path, starting from s_0, in the full state transition graph. For an interleaving sequence w, the set of states it visits is denoted by $states(w)$. If w is finite, the final state reached is denoted by fin_w.

Definition 1. *[6] An* **independence relation** $I \subseteq T \times T$ *is an irreflexive, symmetric relation such that* $(\alpha, \beta) \in I$ *iff* $\forall s \in S$:

- **Enabledness:** $\alpha \in enabled(s) \Rightarrow (\beta \in enabled(s) \Leftrightarrow \beta \in enabled(\alpha(s)))$, *and*
- **Commutativity:** $(\alpha, \beta \in enabled(s)) \Rightarrow (\alpha(\beta(s)) = \beta(\alpha(s)))$.

The enabledness condition states that execution of α does not affect the enabledness of β, and the commutativity condition states that executing α and β in either order results in the same state. The **dependency relation** $D = (T \times T) \setminus I$.

Let $v, w \in T^*$ be two sequences of operations. We say $v \equiv_D w$ iff there exists a sequence of strings $u_0, u_1, ..., u_n$, such that $v = u_0$, $w = v_n$, and for $1 \leq i < n$, $u_i = \nu\alpha\beta\omega$ and $u_{i+1} = \nu\beta\alpha\omega$, for some $\nu, \omega \in T^*$ and $\alpha, \beta \in T$, such that $(\alpha, \beta) \in I$. Informally, $v \equiv_D w$ iff v can be transformed into w by repeatedly commuting adjacent independent operations. Note that if v is an interleaving sequence of a program $\mathcal{P}$, then so is w.

This definition was extended to infinite sequences in [10]. Let $Pref(v)$ denote the set of all finite prefixes of a (finite or infinite) sequence v. We say that $v \preceq_D w$ iff $\forall u \in Pref(v) : \exists y \in Pref(w)$ such that $y \equiv_D z$ and $u \in Pref(z)$. We say $v \equiv_D w$ iff $v \preceq w$ and $w \preceq v$.

It is easy to verify that $\equiv_D$ is an equivalence relation over all interleaving sequences of a program $\mathcal{P}$. Therefore, it partitions the interleaving sequences of $\mathcal{P}$ into equivalence classes called *traces*. Thus, we can denote a trace σ by $[v]$, where v is any member sequence of σ.

We define $States(\sigma) \overset{def}{\equiv} \bigcup_{v \in \sigma} states(v)$. Informally, $States(\sigma)$ includes all the states visited by any interleaving sequence of σ. Note that every interleaving sequence of a trace consists of the same events. The notation σ_E will denote a

[2] Alternatively, a set of initial states may be considered.

trace with E as its set of events. For finite v and w, if $v \equiv_D w$, then $fin_v = fin_w$. So, every finite trace σ has a unique final state, fin_σ.

Mazurkiewicz [6] showed that every trace corresponds to a partial order, such that every linearization of the partial order is an interleaving sequence of the trace and vice-versa. The partially-ordered set (poset) corresponding to a trace σ_E is given by $(E, \rightarrow)$, where the relation $\rightarrow$ corresponds to Lamport's *happened-before* (causality) relation [11]:

Definition 2. *The happened-before relation $\rightarrow$ on a trace $\sigma_E = [w]$ is the smallest transitive relation that satisfies:*

$$(\alpha, \beta) \in D \wedge (w = u\alpha v\beta w') \Rightarrow \alpha \rightarrow \beta$$

where $\alpha, \beta \in E$, and $u, v \in T^*$, $w' \in T^* \cup T^\omega$.

In the rest of this paper, the term "trace" will be used to denote the corresponding partial order. An **ideal** of a trace σ_E is any downward-closed subset of the poset $(E, \rightarrow)$. Formally, any set $G \subseteq E$ is an **ideal** of σ_E if for any $e, f \in E$, $(f \in G) \wedge (e \rightarrow f) \Rightarrow (e \in G)$. We denote the set of all ideals of σ by $I(\sigma)$. An ideal G of a trace σ_E is itself a trace: $\sigma_G = (G, \rightarrow)$. It is well-known [11] that every state in $States(\sigma_E)$ corresponds to fin_{σ_G} for some order ideal G of σ_E, and vice-versa.

3 Traces and Lattices

A well-known result in lattice theory [12] states that the set of all ideals of a poset forms a distributive lattice under the subset relation. For *finite* traces, the corresponding lattice of ideals is also finite.

Definition 3 (Join-Irreducible Element). *An element $x \in L$ is join-irreducible if:*

1. *$x \neq 0$, and*
2. *$\forall\, a, b \in L : x = a \sqcup b \implies (x = a) \vee (x = b)$.*

Here, $\sqcap$ is the *meet* operator, $\sqcup$ is the *join* operator, and 0 refers to the zero element of L. Pictorially, in a finite lattice, an element is join-irreducible iff it has exactly one incoming edge. Let $J(L)$ denote the set of join-irreducible elements of a lattice L.

Predicate filtering is based upon the following theorem by Birkhoff [12]:

Theorem 1. [Birkhoff's Representation Theorem for Finite Distributive Lattices] *Let L be a finite distributive lattice. Then, the map $f : L \rightarrow I(J(L))$ defined by*

$$f(a) = \{x \in J(L) | x \leq a\}$$

is an isomorphism of L onto $I(J(L))$. Dually, let P be a finite poset. Then the map $g : P \rightarrow J(I(P))$ defined by

$$g(a) = \{x \in P | x \leq a\}$$

is an isomorphism of P onto $J(I(P))$.

Informally, a finite distributive lattice L is isomorphic to the lattice of ideals of $J(L)$. Conversely, a poset P is isomorphic to the join-irreducible elements of the lattice $I(P)$.

From the above discussion, it is clear that a trace σ_E is just a *compact* representation of the finite distributive lattice $I(\sigma_E)$. We say *compact* because the number of join-irreducible elements of a lattice is usually exponentially smaller than the *total* number of elements in the lattice. For a finite distributive lattice, the number of join-irreducible elements is exactly equal to the length of the longest chain in the lattice [12], which in this case is bounded by $|E|$. The total number of elements in the lattice itself is bounded by $2^{|E|}$.

4 Finite trace covers

Clearly, any program can be decomposed into a set of traces, which together cover the entire reachable state space of the program. We call such a set of traces a "trace cover".

Definition 4. *A set of traces Δ of a program $P = (S, T, s_0)$ is called a* **trace cover** *iff for every reachable state $s \in S$, there exists a trace $\sigma \in \Delta$ such that $s \in States(\sigma)$.*

Every *finite-state* program can be represented by a *finite trace cover*, that is, a trace cover that consists of a finite set of traces (posets), each of finite length. In [5], we presented an algorithm for constructing such a finite trace cover.

Given a general non-temporal predicate ψ and a trace σ_E, the problem of determining whether any state of σ_E satisfies ψ is NP-complete in $|E|$[8], hence NP-complete for finite trace covers. In [5], we identified two classes of predicates for which reachability could be determined in polynomial time in the size of the finite trace cover.

4.1 Linear predicates

Linear predicates were first introduced in [8], along with a polynomial time algorithm to detect them on a trace. Observe that, if G and H are two ideals of a trace, then so is $G \cap H$.

Definition 5. *A predicate ψ is said to be* **meet-closed** *in a trace σ_E iff for every pair of ideals G and H of σ_E:*

$$(fin_{\sigma_G} \models \psi \wedge fin_{\sigma_H} \models \psi) \Rightarrow (fin_{\sigma_{G \cap H}} \models \psi)$$

A *local predicate* is a predicate that is defined using only local variables from a single process, *e. g.*, $x < 5$, where x is a local variable on some process. Local predicates are meet-closed. If ψ_1 and ψ_2 are meet-closed, then so is $\psi_1 \wedge \psi_2$.

Definition 6. *Let G be an ideal of σ_E such that $fin_{\sigma_G} \not\models \psi$. We say that an event e is* **crucial** *w.r.t. ψ at G if and only if, for all ideals H of σ_E:*

$$(G \subseteq H) \wedge (fin_{\sigma_H} \models \psi) \Rightarrow e \in H \setminus G$$

That is, any state H that can be reached *from* G cannot satisfy ψ unless a crucial event e has been executed along the path from G to H. The following theorem is proved in [8, 5].

Theorem 2. *Let G be any ideal of σ_E, and ψ be a predicate such that $fin_{\sigma_G} \not\models \psi$. If ψ is meet-closed in σ_E, then there exists a crucial event w.r.t. psi at G.*

Definition 7. *A predicate ψ is* **linear** *in a trace σ_E iff ψ is meet-closed in σ_E, and a crucial event can be identified in $O(|E|^k)$ time, for some constant $k \geq 0$.*

As shown in [5], for a linear predicate ψ, $EF(\psi)$ can be detected for a trace σ_E in $O(C.|E|)$ time, where C is the time taken to determine the crucial event. A conjunction of local predicates, $l_1 \wedge l_2 \wedge ... \wedge l_n$, is linear, because at each state in which the predicate is not satisfied, there exists at least one conjunct, say l_i that evaluates to false. Clearly, if l_i is a local variable on process P_i, then the event from P_i in *enabled*(G) is a crucial event.

4.2 Bounded-sum predicates

Another useful class of predicates are those of the form $x_1 + x_2 + ... + x_n > k$, where the x_i are local variables that can only take a value of either 0 or 1, and k is a constant. We call such predicates **bounded-sum** predicates. Bounded-sum predicates can be used to detect mutual exclusion violation ($EF(\sum_i incs_i > 1)$), or to detect if there are more than k copies of a k-licensed software in use at once ($EF(\sum_i in\_use_i > k)$). The problem of detecting $EF(\varphi)$ on a trace σ_E, for a bounded-sum predicate φ, can be reduced to the problem of computing the width of a poset[5].

5 Predicate filters

Predicate filters are best introduced through an example. Figures 1(a) and (b), respectively, show a trace σ and its corresponding lattice of ideals, $I(\sigma)$. The label of each element in the lattice is a set containing the maximal event from each process contained in the corresponding ideal. For example, the label of the ideal $\{e_1, f_1, f_2, g_1\}$ is $\{e_1, f_2, g_1\}$.

Let ϕ be a given predicate. The shaded elements of the lattice $I(\sigma)$ correspond to the states that satisfy ϕ. Figure 1(c) shows the *smallest sublattice* of $I(\sigma)$ that contains *all* the shaded elements. We denote this sublattice by L_ϕ. Note that, in order to make L_ϕ a *sublattice* of $I(\sigma)$, we need to include some non-shaded elements (*i.e.*, states that do not satisfy ϕ). Now, every sublattice of a distributive lattice is also distributive [12], so we can apply Birkhoff's Representation Theorem (Theorem 1) to L_ϕ. Figure 1(d) shows the poset induced by the join-irreducible elements of L_ϕ. We call this poset the *filtrate* of σ with respect to ϕ.

As seen in Figure 1(d), the filtrate is a poset in which each element is a *subset* of the events of the original trace. This reflects the fact that any state

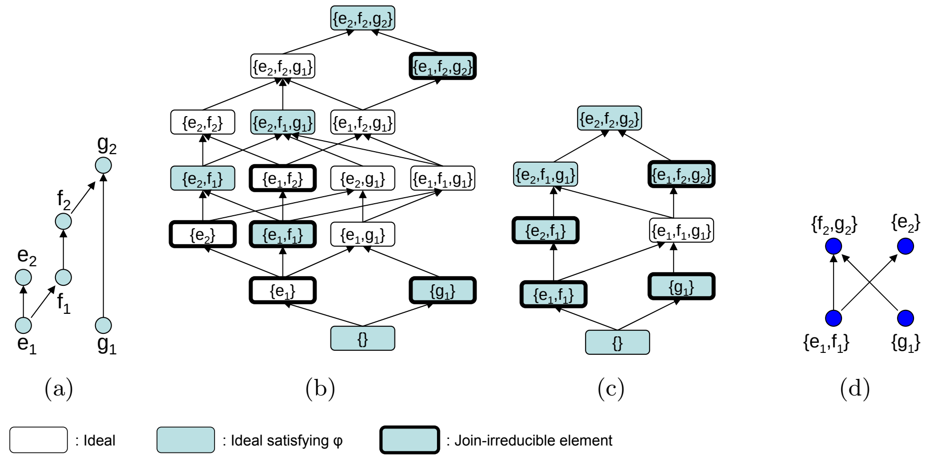

Fig. 1. Predicate filters. (a) A trace, σ, (b) its lattice of ideals, $I(\sigma)$, (c) the sublattice L_ϕ induced by the states that satisfy ϕ, and (d) the filtrate of σ w.r.t. ϕ.

that satisfies the given predicate ϕ either contains all the events corresponding to a subset, or none of them. Thus, these events are merged into a single atomic event in the filtrate.

Definition 8. *The* **filtrate** *of a trace σ w.r.t. a predicate ϕ is the trace σ' such that $I(\sigma')$ is the smallest sublattice of $I(\sigma)$ that contains all the ideals of σ that satisfy ϕ.*

The intersection of any two sublattices of a lattice L is also a sublattice of L [12]. Consequently, the *smallest* sublattice L_ϕ is unique and well-defined.

Theorem 3. *The filtrate of any trace σ w.r.t. any predicate ϕ is unique and well-defined.*

In order to avoid state space explosion while deriving the filtrate of a trace, one has to avoid construction of the lattices L and L_ϕ, because they could be prohibitively large. Thus, we need a way to directly compute the filtrate from the poset representation of a trace.

5.1 Constructing the filtrate

A poset is usually represented by a Hasse diagram [12], which is essentially a directed acyclic graph (DAG). In this section, we will find it useful to view a poset as a directed graph.

The notion of ideals can be extended to directed graphs in a straightforward manner. A subset of vertices, H, of a directed graph P is an *ideal* if whenever H contains a vertex v and (u, v) is an edge in the graph P, then H also contains u. Observe that an ideal of P either contains all vertices in a strongly connected component, or none of them. Let $\mathcal{I}(P)$ denote the set of ideals of a directed graph P. The following theorem is a slight generalization of the result in lattice theory that the set of ideals of a poset forms a distributive lattice [12].

Theorem 4. *[9] Given a directed graph P, $\mathcal{I}(P)$ forms a distributive lattice under the subset relation.*

Observe that the empty set and the set of all vertices trivially belong to $\mathcal{I}(P)$. We call them *trivial* ideals. We can construct a graph P corresponding to a trace σ such that there is one-to-one correspondence between all ideals of σ and all *nontrivial* ideals of P. To construct P, we add two additional vertices to the Hasse diagram representation of σ, $\perp$ and $\top$, where $\perp$ is the "smallest" vertex and $\top$ is the "largest" vertex (*i.e.*, there is a path from $\perp$ to every vertex and a path from every vertex to $\top$). An example of such a transformation is shown in Figure 2.

Clearly, any nontrivial ideal of P will contain $\perp$ and not contain $\top$. As a result, every ideal of σ is a nontrivial ideal of P and vice versa.

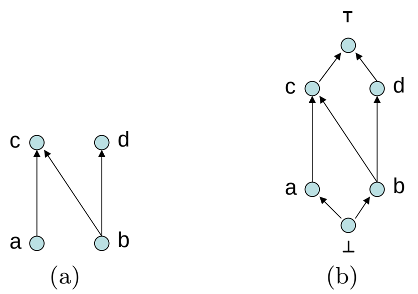

Fig. 2. (a) A trace σ, and (b) its corresponding directed graph, P.

It is obvious that adding edges to a directed graph P can only *reduce* the number of its ideals. Thus, any directed graph P' obtained by adding edges to P will produce a sublattice $\mathcal{I}(P')$ of $\mathcal{I}(P)$. The following theorem, from [13], states that *every* sublattice of $\mathcal{I}(P)$ can be derived by adding edges to P.

Theorem 5. *Let L be the finite distributive lattice of ideals generated by the graph P. Let L' be any sublattice of L. Then, there exists a graph P' that can be obtained by adding edges to P that generates L'.*

The algorithm shown in Figure 3 [13] computes the filtrate directly from the directed graph representation of a trace. It takes as input a directed graph P, derived from a trace σ, and a predicate ϕ. The algorithm constructs the filtrate by adding edges to the graph P. In line (3), a graph R in initialized to P. In rest of the function, edges are added to R, which is finally returned. The **filtrate** of P with respect to ϕ is simply the poset (DAG) obtained by collapsing the strongly-connected components of R into a single node.

```
(1)  graph function computeFiltrate($\phi$: predicate, $P$: directed graph)
(2)  var
(3)      $R$: directed graph initialized to $P$;
(4)  begin
(5)      for every pair of vertices $e$, $f$ in $P$ do
(6)          $Q := P$ with the additional edges $(f, \bot)$ and $(\top, e)$;
(7)          if $detect(Q, \phi)$ is false
(8)              add edge $(e, f)$ to $R$;
(9)      endfor
(10)     return $R$;
(11) end;
```

Fig. 3. Algorithm to compute the filtrate with respect to a predicate ϕ

For each pair of vertices e and f in the graph P, the algorithm constructs a graph Q from P by adding two additional edges: one from f to $\bot$, and the other from $\top$ to e (line (6)). Any *non-trivial* ideal of Q cannot contain $\top$, so it cannot contain e. On the other hand, it must contain $\bot$, hence must contain f. Therefore, every non-trivial ideal of Q must contain f, but must not contain e. The algorithm $detect(Q, \phi)$ checks whether there exists any non-trivial ideal of Q that satisfies ϕ.

If $detect(Q, \phi)$ returns *false*, that means no **satisfying** ideal of the trace σ contains f but not e. Therefore, adding an edge from e to f in P will not eliminate any satisfying ideals, while it *will* create a sublattice of $I(\sigma)$. We continue this procedure for all pairs of vertices. With each added edge, we produce an even *smaller* sublattice of $I(\sigma)$, all the time retaining every satisfying ideal of $I(\sigma)$.

Theorem 6. *[13] Let P be a directed graph. Let R be the directed graph output by the algorithm in Figure 3 for a predicate ϕ. Then R is the filtrate of P w.r.t. ϕ.*

Proof. Let $\mathcal{I}(P, \phi)$ denote the set of ideals of P that satisfy ϕ. We first show that $\mathcal{I}(R) \supseteq \mathcal{I}(P, \phi)$. Adding an edge (e, f) in R eliminates only those ideals of P that contain f but do not contain e. But, all those ideals do not satisfy ϕ because the edge (e, f) is added only when $detect(Q, \phi)$ is false. Thus, all the ideals of P that satisfy ϕ are also the ideals of R.

Next, we show that $\mathcal{I}(R)$ is the smallest sublattice of $\mathcal{I}(P)$ that includes $\mathcal{I}(P, \phi)$. Let M be a graph such that $\mathcal{I}(M) \supseteq \mathcal{I}(P, \phi)$. Assume, if possible, $\mathcal{I}(M)$ is strictly smaller than $\mathcal{I}(R)$. This implies that there exists two vertices e and f such that there is an edge from e to f in M but not in R. Since R is output by the algorithm, $detect(Q, \phi)$ is true in line (7); otherwise, an edge would have been added from e to f. But, this means that there exists an ideal in P which includes f, does not include e, and satisfies ϕ. This ideal cannot be in $\mathcal{I}(M)$ due to the edge from e to f, contradicting our assumption that $\mathcal{I}(P, \phi) \subseteq \mathcal{I}(M)$. $\square$

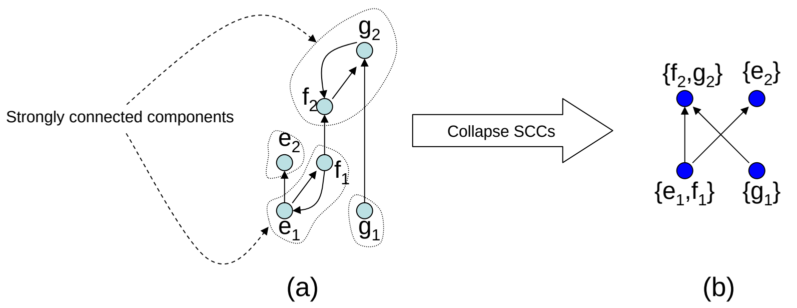

Fig. 4. Directed graph returned by *computeFiltrate*() for the example in Figure 1.

Figure 4 shows the directed graph R returned by *computeFiltrate*() for the example in Figure 1. By collapsing the strongly-connected components of the graph in Figure 4, we obtain the filtrate shown in Figure 1(d).

Assuming $detect(Q, \phi)$ takes $O(t)$ time, the function *computeFiltrate*() has a time complexity of $O(t.|E|^2)$. Thus, if ϕ can be detected in polynomial time on a trace, then the filtrate w.r.t. ϕ can be computed in polynomial time.

The converse is also true. That is, if we can compute the filtrate of a trace with respect to a predicate in polynomial time, then we can also *detect* the predicate in polynomial time on the trace. To see why this is true, observe that if the lattice of ideals of a trace contains no states that satisfy a predicate ϕ, then the smallest sublattice that includes all satisfying states is the *empty* sublattice. That is, the corresponding filtrate will contain no non-trivial ideals. In this case, *computeFiltrate*() would return a graph R with exactly one strongly connected component, which contains all the events of P. The following theorem is formally proved in [14]:

Theorem 7. *The filtrate of a trace σ with respect to a predicate ϕ can be computed in polynomial time if and only if there exists a polynomial time algorithm to determine whether there exists a state of σ that satisfies ϕ.*

A filtrate is said to be **pure** if each of its ideals satisfies ϕ. Clearly, if the ideals of σ that satisfy ϕ themselves form a sublattice of $I(\sigma)$, then the corresponding filtrate will be pure. If a predicate ϕ is both meet- and join-closed in a trace, then the filtrate w.r.t. ϕ will be pure [9].

6 State space reduction using predicate filters

Figure 5 demonstrates that the filtrate of a trace can be represented by a program. The figure shows the filtrate as a PROMELA program. PROMELA is the specification language for the model checker SPIN [15].

Let ψ be a non-temporal predicate of the form $\psi = \psi_1 \wedge \psi_2$, where ψ_1 belongs to one of the efficient predicate classes presented in Section 4, and ψ_2 does not

```
init {
    byte $e_1 f_1\_done = 0$, $g_1\_done = 0$, $e_2\_done = 0$, $f_2 g_2\_done = 0$;
    byte $start = 1$;
    do
        :: atomic { ($start$ && !$e_1 f_1\_done$) $\rightarrow$ $e_1$; $f_1$; $e_1 f_1\_done = 1$}
        :: atomic { ($start$ && !$g_1\_done$) $\rightarrow$ $g_1$; $g_1\_done = 1$}
        :: atomic {($e_1 f_1\_done$ && !$e_2\_done$) $\rightarrow$ $e_2$; $e_2\_done = 1$ }
        :: atomic {($e_1 f_1\_done$ && $g_1\_done$ && !$f_2 g_2\_done$) $\rightarrow$ $f_2$; $g_2$}
        :: else $\rightarrow$ break
    od
}
```

Fig. 5. Equivalent PROMELA program for the filtrate in Figure 1(d)

belong to any known efficient class. We are required to determine whether any reachable state of a given program $\mathcal{P}$ satisfies ψ. We can use predicate filtering to make this task easier, as follows.

First, we construct a finite trace cover for $\mathcal{P}$, using the algorithm in [5]. Let Δ be the set of traces in the cover. For each trace $\sigma \in \Delta$, we compute the filtrate of σ w.r.t. ψ_1. This can be done in polynomial time in the size of the trace cover, as ψ_1 belongs to an efficient predicate class. Some of these filtrates may be empty, that is, no state of the corresponding trace may satisfy ψ_1. Then, we can check for ψ_2 in each non-empty filtrate by using other model checking approaches, such as explicit state construction. For example, if each non-empty filtrate is represented by a PROMELA program, such as the one shown in Figure 5, then we can invoke SPIN to detect ψ_2. Note that ψ_1 is a weaker predicate than ψ, so the filtrate will contain all the states that satisfy ψ.

6.1 Case Study: Leader Election Protocol

We implemented our algorithms from [5], and the filtering algorithms presented here, by modifying SPIN. Our implementation, called TC-SPIN ("trace cover" SPIN) takes an input PROMELA program, uses SPIN's parser and dependency relation calculations to convert the input program into a finite trace cover. Then, we filter each trace with respect to a user-specified predicate and output a PROMELA program for each non-empty filtrate.

For our experiments, we used a PROMELA specification of Dolev, Klawe and Rodeh's [16] leader election protocol, with random initialization of processes. This PROMELA specification is part of the standard SPIN distribution. We added two local variables to each process: *know_leader*, which is set to 1 when the process knows the identity of the leader, and *leader_id*, which is set to the process id of the leader. The property being validated is that, once a leader is elected, every process is in agreement about the leader's identity:

$$\neg EF((\bigwedge_i know\_leader_i) \wedge (\nexists j : leader\_id_j \neq leader\_id_{(j+1)\%N}))$$

In the SPIN verification run, the property is specified by means of an assert statement in a separate monitor process.

TC-SPIN executes in two passes - in Pass 1, it creates the finite trace cover and computes the filtrate of each trace w.r.t. the predicate $\bigwedge_i know\_leader_i$. This predicate is linear (a conjunction of local variables), and hence the filtrate can be computed efficiently. Each non-empty filtrate is written out as a PROMELA program. In Pass 2, SPIN is called on each filtrate. The property being verified in Pass 2 is $AG(\forall i : leader_i == leader_{(i+1)\%N})$. This property is specified by means of an assert statement in a separate monitor process. Table 1 shows the number of states constructed (stored) by SPIN vs. TC-SPIN during verification. For TC-SPIN, the number of states in Pass 2 is the sum of all the states generated during verification of all the filtrates. In this example, the filtrates only created about 15 states each. The number of filtrates itself increased with the value of N, because the amount of non-deterministic choice in the program was directly proportional to N. The number of states in Pass 1 is the total number of states generated (stored) during construction of the finite trace cover **and** the filtrate computation. Further details of the experiment can be obtained from: `http://maple.ece.utexas.edu`.

	SPIN, P.O. reduction	TC-SPIN - # states		
# processes	Total # states	Pass 1	Pass 2	Total
4	3985	2465	345	2810
5	47727	16721	1785	18506
6	408091	125755	9630	135385

Table 1. Number of states constructed during verification of the leader election protocol.

The time taken by SPIN vs. TC-SPIN was comparable for these examples. However, our focus in these experiments was on the amount of state space reduction achieved, because memory consumption is usually the larger concern during verification.

7 Concluding Remarks

Predicate filtering is a useful state space reduction tool. In addition, it can be used as a debugging aid. For example, consider a classic deadlock problem in which each process is waiting on another process in a circular dependency loop. A programmer can use the filter $\bigwedge_i waiting_i$, to isolate all the states in which every process is in the waiting state. Incidentally, this predicate is meet- and join-closed, so the filtrate is pure. Filtering allows the programmer to develop a better understanding of the sequence of events that leads to error states. Note that the filtration process captures *every* error state, whereas a traditional model checker only reports *one* error state in its counterexample trace.

In our future work, we hope to study the usefulness of predicate filtering as a debugging aid. Another direction of research is to include the ability to verify temporal properties using predicate filtering.

References

1. Peled, D.: All from One, One for All: on Model Checking Using Representatives. Volume 697 of Lecture Notes in Computer Science. Springer-Verlag (1993)
2. Valmari, A.: Stubborn Sets for Reduced State Space Generation. Volume 483 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, Germany (1990)
3. Godefroid, P.: Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem. Volume 1032. Springer-Verlag Inc., New York, NY, USA (1996)
4. Gerth, R., Kuiper, R., Peled, D., Penczek, W.: A partial order approach to branching time logic model checking. In: Proceedings of the Third Israel Symposium on the Theory of Computing and Systems (ISTCS'95), Tel Aviv, Israel, January 4-6, 1995. (1995)
5. Kashyap, S., Garg, V.K.: Exploiting predicate structure for efficient reachability detection. In: ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, ACM Press (2005) 4–13
6. Mazurkiewicz, A.W.: Basic notions of trace theory. In: Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, London, UK, Springer-Verlag (1989) 285–363
7. Howell, R.R., Rosier, L.E.: Completeness results for conflict-free vector replacement systems. J. Comput. Syst. Sci. **37**(3) (1988) 349–366
8. Chase, C.M., Garg, V.K.: Efficient detection of restricted classes of global predicates. In: WDAG '95: Proceedings of the 9th International Workshop on Distributed Algorithms, London, UK, Springer-Verlag (1995) 303–317
9. Mittal, N., Garg, V.K.: Computation slicing: Techniques and theory. In: DISC '01: Proceedings of the 15th International Conference on Distributed Computing, London, UK, Springer-Verlag (2001) 78–92
10. Kwiatkowska, M.Z.: Event fairness and non-interleaving concurrency. Formal Aspects of Computing **1** (1989) 213–228
11. Lamport, L.: Time, clock and the ordering of events in a distributed system. Communications of the ACM (CACM) **21**(7) (1978) 558–565
12. Davey, B., Priestly, H.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (1990)
13. Garg, V.K.: Algorithmic combinatorics based on slicing posets. In: FSTTCS '02: Proceedings of the 22nd Conference Kanpur on Foundations of Software Technology and Theoretical Computer Science, London, UK, Springer-Verlag (2002) 169–181
14. Mittal, N., Sen, A., Garg, V.K., Atreya, R.: Finding satisfying global states: All for one and one for all. In: IPDPS 2004: Proceedings of the 18th International Parallel and Distributed Processing Symposium, Santa Fe, New Mexico, USA (2004)
15. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
16. Dolev, D., Klawe, M., Rodeh, M.: An $O(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. Journal of Algorithms **3** (1982) 245–260