Cover Algorithms and Their Combination

Sumit Gulwani and Madan Musuvathi

Microsoft Research, Redmond, WA, 98052
{sumitg,madanm}@microsoft.com

Abstract. This paper defines the cover of a formula ϕ with respect to a set of variables V in theory T to be the strongest quantifier-free formula that is implied by $\exists V : \phi$ in theory T. Cover exists for several useful theories, including those that do not admit quantifier elimination. This paper describes cover algorithms for the theories of uninterpreted functions and linear arithmetic. In addition, the paper provides a combination algorithm to combine the cover operations for theories that satisfy some general condition. This combination algorithm can be used to compute the cover a formula in the combined theory of uninterpreted functions and linear arithmetic. This paper motivates the study of cover by describing its applications in program analysis and verification techniques, like symbolic model checking and abstract interpretation.

1 Introduction

Existential quantifier elimination is a core primitive used in several program analysis and verification techniques. Given a quantifier-free formula ϕ and a set of variables V, existentially quantifying away V involves computing a quantifier-free formula that is logically equivalent to $\exists V : \phi$. This operation is useful in practice to eliminate variables that are no longer necessary from a formula. For instance, the image computation in symbolic model checking [14] involves computing the quantifier-free formula equivalent to $\exists V : R(V) \wedge T(V, V')$. Here, $R(V)$ represents the current set of reachable states and $T(V, V')$ represents the transition relation between the current values of the state variables V and their new values V'.

Existential quantifier elimination can be performed, albeit with exponential complexity, for propositional formulas. However, this operation is not defined for formulas containing interpreted symbols from certain theories. For example, consider the formula $F(x) = 0$ in the theory of uninterpreted functions. There is no quantifier-free formula that is equivalent to $\exists x : F(x) = 0$ as it is not possible to state that 0 is in the range of function F without using quantifiers. This limits the application of techniques like symbolic model checking to systems described by formulas in these theories.

To address this problem, we introduce the notion of *cover*. Given a quantifier-free formula ϕ containing interpreted symbols from theory T and a set of variables V, we define $\mathbb{C}_T V : \phi$ (called the cover of ϕ with respect to V in T) as the strongest quantifier-free formula in T that is implied by $\exists V : \phi$. Formally, the cover operation satisfies the following constraints.

S. Drossopoulou (Ed.): ESOP 2008, LNCS 4960, pp. 193–207, 2008.

$$(\exists V : \phi) \quad \Rightarrow_T \quad (\mathbb{C}_T V : \phi)$$
$$((\exists V : \phi) \Rightarrow_T \gamma) \quad \text{iff} \quad ((\mathbb{C}_T V : \phi) \Rightarrow_T \gamma)\,, \text{ for all quantifier-free formulas } \gamma$$

When the theory T is obvious from context, we drop the subscript T from the notation and refer to the cover simply as $\mathbb{C}V : \phi$.

Intuitively, applying the cover operation on a formula with respect to V eliminates all variables in V from the formula. However, the resulting formula only retains *quantifier-free* facts pertaining to other variables in the formula. For an example, let ϕ be the formula $y = \texttt{Mem}(a + x) - \texttt{Mem}(b + x)$, where Mem is an uninterpreted function. Using cover to eliminate the variable x, we get

$$(\mathbb{C}x : y = \texttt{Mem}(a + x) - \texttt{Mem}(b + x)) \quad \equiv \quad (a = b \Rightarrow y = 0)$$

Note that $\exists x : \phi$ implies the right hand side of the above equation. The results in this paper show that this is the most precise quantifier-free formula that is implied by ϕ and that does not involve x. Example 3 in Section 4 provides an algorithm to compute the cover of this formula, and Section 2.2 describes an application that requires computing the cover of such formulas. Finally, the reader should also note that applying cover does not retain *quantified* facts. For example, $\mathbb{C}x : \phi$ does not imply the fact $(\forall x : \texttt{Mem}(a + x) = \texttt{Mem}(b + x)) \Rightarrow y = 0$, while $\exists x : \phi$ does.

This distinguishing fact of cover allows us to define this operation even for theories that do not admit existential quantifier elimination. In Section 3, we describe the cover algorithm for the theory of uninterpreted functions. Note that cover is trivially defined for propositional formulas and theory of linear arithmetic, as cover, by definition, reduces to existential quantifier elimination when it exists.

In Section 4, we present a *combination* algorithm for computing cover for union of two theories that individually support cover operations and satisfy some general condition. Our combination algorithm is based on extension of Nelson-Oppen methodology for combining decision procedures [16]. However, in our combination framework, we also need to exchange *conditional* variable equalities (of the form $\gamma \Rightarrow v_1 = v_2$) and variable-term equalities (of the form $\gamma \Rightarrow v = t$) between component theories. Our combination algorithm works for theories that are convex, stably infinite, disjoint, and have a finite set of *simple terms* (Definition 1 in Section 4). The theories of linear arithmetic and uninterpreted functions, for example, satisfy these constraints.

We also describe how the cover operation can be used in program analysis and verification techniques that otherwise depend on existential quantifier elimination. In particular, this paper presents a modified symbolic model checking algorithm (Section 2.1) using the cover operation in the image computation step. This new algorithm can be used to reason about transition systems involving operations from the rich set of theories for which the cover operation is defined. Moreover, when the transition system can be described using quantifier-free formulas, we show that the symbolic model checking algorithm using cover is not only sound, but also *precise* (Theorem 1). In other words, when the checking algorithm terminates, any error reported is guaranteed to be a real error in

the system. This is in stark contrast with other over-approximation based techniques [1,3,15], which only guarantee soundness. Precision is very important for *falsification* techniques. A similar application is in performing abstract interpretation of programs over abstractions whose elements are quantifier-free formulas describing program states.

In summary, this paper has the following main contributions.

- We introduce the notion of cover as the most precise quantifier-free over-approximation to existential quantifier elimination. We study this operation and present its useful properties.
- As a practical application, we present a new symbolic model checking algorithm using cover. This algorithm is both sound and *precise*, and can be used to reason about transition systems described using formulas in a rich set of theories.
- We show how to do a precise analysis of programs by performing abstract interpretation over abstract domains that describe program states using quantifier-free formulas.
- We show that cover can be computed for the theory of uninterpreted functions.
- We present an extension to the Nelson-Oppen combination framework that can be used to combine the cover operation of theories satisfying a general condition. We show that useful theories such as the theory of uninterpreted functions and linear arithmetic satisfy these conditions.

2 Applications of Cover

Before presenting cover algorithms for some theories, and a methodology for combining cover algorithms in the following sections, we first motivate the study of cover by describing some useful applications for the cover operation.

2.1 Symbolic Model Checking

Our main motivation for cover is to apply symbolic model checking to reason about transition systems that involve rich operations from the theory of uninterpreted functions, which naturally arise in program analysis and software verification.

A transition system can be described by the tuple $(V, I(V), T(V_{old}, V_{new}), E(V))$, where V represents the set of state variables, $I(V)$ is a formula describing the set of initial states, $T(V_{old}, V_{new})$ is a formula describing the transition relation between the old values V_{old} and new values V_{new} of the variables in V, and $E(V)$ is a formula describing the set of error states. For clarity, if $\phi(V)$ is a formula with variables from V, we will use $\phi(V')$ to be the formula obtained from ϕ by renaming each variable in V with the corresponding variable in V'.

Given a transition system, the symbolic model checking algorithm computes the set of reachable states $R(V)$ iteratively as follows.

$$R_0(V) \equiv I(V)$$
$$R_i(V) \equiv R_{i-1}(V) \vee (\exists V_{old} \colon R_{i-1}(V_{old}) \wedge T(V_{old}, V)) \quad \text{for } i > 0$$

This iteration reaches a fix point at n if and only if $R_n(V) \Rightarrow R_{n-1}(V)$. At this point, $R_n(V)$ is an inductive invariant of the transition system. Also, if $R_n(V) \Rightarrow \neg E(V)$ then the system does not reach an error state.

A transition system $(V, I(V), T(V_{old}, V_{new}), E(V))$ is quantifier-free when the formulas $I(V), T(V_{old}, V_{new})$, and $E(V)$ are all quantifier-free. In practice, transition systems arising from many software verification applications are quantifier-free. For such systems, we propose a new symbolic model checking algorithm that uses the cover operation instead of existential quantification. Also, we show that this new algorithm is sound and precise. Moreover, this algorithm terminates whenever the original model checking algorithm terminates.

In the discussion below, we assume that the transition system uses operations from a theory T, such as the union of the theory of reals and uninterpreted functions. We assume that the cover operations are performed with respect to this theory. (See Section 3 for the actual cover algorithms.)

The symbolic model checking algorithm using cover is as follows.
SMC-Cover Algorithm:

$$CR_0(V) \equiv I(V)$$
$$CR_i(V) \equiv CR_{i-1}(V) \vee (\mathbb{C}V_{old} \colon CR_{i-1}(V_{old}) \wedge T(V_{old}, V)) \quad \text{for } i > 0$$

In the equations above, $CR_i(V)$ determines the set of reachable states determined using the cover operation after i iterations. The fix point is reached, as before, at point n when $CR_n(V) \Rightarrow CR_{n-1}(V)$.

Lemma 1. *Given a quantifier-free transition system* $(V, I(V), T(V_{old}, V_{new}), E(V))$, $CR_n(V) \Rightarrow \phi$ *if and only if* $R_n(V) \Rightarrow \phi$ *for all quantifier-free formulas* ϕ.

Proof. The proof is by induction. The base case is trivial as $CR_0(V) \equiv I(V) \equiv R_0(V)$. For the induction, assume the lemma holds for all iterations up to $n-1$. Note, that by definition

$$R_n(V) \equiv \exists V_{old} \colon R_{n-1}(V_{old}) \wedge T(V_{old}, V) \vee R_{n-1}(V)$$

$$CR_n(V) \equiv \mathbb{C}V_{old} \colon CR_{n-1}(V_{old}) \wedge T(V_{old}, V) \vee CR_{n-1}(V)$$

Consider a quantifier-free formula ϕ that does not contain, without loss of generality, variables from V_{old}.[1] Now, if $R_n(V) \Rightarrow \phi$, then the following are true

$$R_{n-1}(V) \Rightarrow \phi$$
$$\exists V_{old} \colon R_{n-1}(V_{old}) \wedge T(V_{old}, V) \Rightarrow \phi$$

[1] The variables from V_{old}, if present, can be renamed.

From the first equation, we have $CR_{n-1}(V) \Rightarrow \phi$ by induction. Moreover from the second equation, we have

$$R_{n-1}(V_{old}) \wedge T(V_{old}, V) \Rightarrow \phi \text{ as } \phi \text{ does not contain variables in } V_{old}$$
$$R_{n-1}(V_{old}) \Rightarrow (T(V_{old}, V) \Rightarrow \phi)$$
$$CR_{n-1}(V_{old}) \Rightarrow (T(V_{old}, V) \Rightarrow \phi) \text{ by induction, as } T(V_{old},V) \text{ is quantifier-free}$$
$$CR_{n-1}(V_{old}) \wedge T(V_{old}, V) \Rightarrow \phi$$
$$\mathbb{C}V_{old}: CR_{n-1}(V_{old}) \wedge T(V_{old},V) \Rightarrow \phi \text{ by definition}$$

Thus, we have $CR_n(V) \Rightarrow \phi$, proving the *if* direction of the lemma. Proving the other direction is similar and follows from the property that cover over-approximates existential quantification.

Using Lemma 1 and the following properties of cover, we can prove the desired result stated in Theorem 1.

Property 1. $\mathbb{C}V : \mathbb{C}W : \phi(V, W) \equiv \mathbb{C}V, W : \phi(V, W)$

Property 2. $\mathbb{C}V : \exists W : \phi(V, W) \equiv \mathbb{C}V : \mathbb{C}W : \phi(V, W)$

Theorem 1. *Given a transition system $(V, I(V), T(V_{old}, V_{new}), E(V))$, where both $T(V_{old}, V_{new})$ and $E(V)$ are quantifier-free, then the symbolic model checking algorithm using cover is sound and precise.*

Proof. The proof follows from Lemma 1. Since $E(V)$ is quantifier-free, $R_n(V) \Rightarrow \neg E(V)$ if and only if $CR_n(V) \Rightarrow \neg E(V)$. Thus, the symbolic model checking algorithm using cover proves the absence of error whenever the original symbolic model checking algorithm proves the same. Also, when the former algorithm reports an error, the latter reports the error.

Theorem 2. *Given a transition system $(V, I(V), T(V_{old}, V_{new}), E(V))$, where both $T(V_{old}, V_{new})$ and $E(V)$ are quantifier-free, then the symbolic model checking algorithm using cover terminates whenever the symbolic model checking algorithm terminates.*

Proof. Say, the symbolic model checking algorithm terminates at step n, then $R_n(V) \Rightarrow R_{n-1}(V)$. Thus, by Lemma 1, we have $R_n(V) \Rightarrow CR_{n-1}(V)$. Since $CR_{n-1}(V)$ is a quantifier-free formula we have $CR_n(V) \Rightarrow CR_{n-1}(V)$. Thus the symbolic model checking algorithm using cover terminates.

Checking Infinite State Systems. The algorithm mentioned above is, in general, not guaranteed to terminate when the transition system describes an infinite state systems. To guarantee termination, this algorithm has to be combined with appropriate abstraction [15] or *widening* techniques [4] to selectively lose facts regarding the set of reachable states. Designing such algorithms is beyond the scope of this paper. However, the cover operation, as opposed to a less precise approximation to existential quantification, is still useful in this setting because it greatly simplifies the design of subsequent refinement [1,3] algorithms. In particular, refinement needs to be performed only at the 'widen' points where information is lost [6].

```
void foo(int a[], int b[]) {
     int y = 0; int x = ?;
     while(*) { y = y + a[x] - b[x]; x = ?; }
     if (y ≠ 0) { assert(a ≠ b); }
}
```

Fig. 1. An example program whose loop invariant (required to prove the assertion) can be generated using cover operation

2.2 Abstract Interpretation over Precise Abstractions

Abstract Interpretation is a well-known methodology to analyze programs over a given abstraction [4]. An abstract interpreter performs a forward analysis on the program computing invariants (which are elements of the underlying abstract lattice over which the analysis is being performed) at each program point. The invariants are computed at each program point from the invariants at the preceding program points in an iterative manner using appropriate transfer functions.

Most of the abstract interpreters that have been described in literature operate over an abstraction whose elements are usually conjunction of atomic predicates in some theory, e.g., linear arithmetic [5], uninterpreted functions [8,9]. These abstractions cannot reason about disjunctive invariants in programs and there is a loss of precision at join points in programs.

Abstractions whose elements are boolean combinations of atomic predicates in an appropriate theory can reason about disjunctive invariants in programs. The join operation (required to merge information at join points) for such an abstraction is simply disjunction, while the meet operation (required to gather information from conditional nodes) is simply conjunction. However, the strongest postcondition operation (required to compute invariants across assignment nodes) is non-trivial. In fact, it is exactly the cover operation for the underlying theory. Hence, a cover operation for a theory can be used to perform abstract interpretation of programs over an abstraction whose elements are quantifier-free formulas over that theory.

Consider, for example, the program shown in Figure 1. We do not know of any existing abstract interpreter that can prove the assertion in the program. For this, we need to do abstract interpretation over the abstraction of quantifier-free formulas in the combined theory of linear arithmetic and uninterpreted functions. Analyzing the first loop iteration involves computing the strongest postcondition of $y = 0$ with respect to the assignment $y := y + a[x] - b[x]$ (in the abstraction of quantifier-free formulas), which is equivalent to computing $\mathbb{C}x', y' : (y' = 0 \wedge y = y' + \texttt{Mem}(a + x') - \texttt{Mem}(b + x') \wedge x = *)$, where Mem denotes the deference operator and can be regarded as an uninterpreted function. This yields the formula $a = b \Rightarrow y = 0$, which also turns out to be the loop invariant and hence fixed point is reached in the next loop iteration.

Furthermore, the invariant computed at the end of the procedure can be turned into a procedure summary by eliminating the local variables of the procedure, again by using the cover operation. Procedure summaries are very useful in performing a context-sensitive reasoning of a program in a modular fashion.

2.3 Computation of Interpolants

Finally, the cover operation can be used to compute *quantifier-free interpolants*. Let $\phi_1(V_1, V)$ and $\phi_2(V_2, V)$ be quantifier-free formulas, such that ϕ_1 contains variables in $V_1 \cup V$, ϕ_2 contains variables in $V_2 \cup V$, $V_1 \cap V_2 = \emptyset$, and $\phi_1(V_1, V) \Rightarrow \phi_2(V_2, V)$. A quantifier-free interpolant $I(V)$ is a quantifier-free formula that contains only variables from V and satisfies $(\phi_1(V_1, V) \Rightarrow I(V)) \wedge (I(V) \Rightarrow \phi_2(V, V_2))$. We can see that $\mathbb{C}V_1 : \phi_1(V, V_1)$ and $\neg(\mathbb{C}V_2 : \neg\phi_2(V, V_2))$ are (respectively the strongest and weakest) quantifier-free interpolants. Such interpolants have recently been used in fix point computations [15] and to refine abstractions [11].

3 Cover Algorithm for the Theory of Uninterpreted Functions

The cover algorithm for theory of uninterpreted functions is given in Figure 3. The algorithm assumes that there are only binary uninterpreted functions, but it can be easily extended to handle uninterpreted functions of any arity.

Property 3. The cover operation distributes over disjunctions, i.e.,

$$(\mathbb{C}V : (\phi_1 \vee \phi_2)) \quad \equiv \quad (\mathbb{C}V : \phi_1) \vee (\mathbb{C}V : \phi_2)$$

Hence, without loss of any generality, the algorithm assumes that the input formula ϕ is a conjunction of atomic facts, where each atomic fact is either a positive or negative atom.

The reasoning behind the cover algorithm is as follows. Suppose $\phi(U, V) \Rightarrow \gamma(U, W)$ such that $U = \texttt{Vars}(\phi) \cap \texttt{Vars}(\gamma)$, $U \cap V = \emptyset$, and $W \cap U = \emptyset$. We require $\mathbb{C}V : \phi \Rightarrow \gamma$. By Craig's interpolant theorem, there exists a $\delta(U)$ such that $(\phi(U, V) \Rightarrow \delta(U)) \wedge (\delta(U) \Rightarrow \gamma(U, W))$. The fact that one can find a quantifier-free interpolant for formulas in the theory of uninterpreted functions follows from [15]. Without loss of generality, one can represent $\delta(U)$ (in conjunctive normal form) as a conjunction of clauses where each clause is of the form $(s_1 = t_1 \wedge \ldots \wedge s_a = t_a) \Rightarrow (s'_1 = t'_1 \vee \ldots \vee s'_b = t'_b)$, where the terms s_i, t_i, s'_j, t'_j only contain variables from U. Therefore, $\phi(U, V)$ implies each of the clauses individually. Finally, from the convexity of the theory of uninterpreted functions [16], whenever $\phi(U, V)$ implies $(s_1 = t_1 \wedge \ldots \wedge s_a = t_a) \Rightarrow (s'_1 = t'_1 \vee \ldots \vee s'_b = t'_b)$, there exists some $1 \leq i \leq b$ such that $\phi(U, V)$ implies $(s_1 = t_1 \wedge \ldots \wedge s_a = t_a) \Rightarrow s'_i = t'_i$. Lines 12 and 15 in the function $\texttt{ComputeCover}_{\texttt{uf}}(\phi)$ compute all such implied equalities. While there could be infinite such implied equalities, one only needs to consider equalities of the form $s_j = t_j$, $1 \leq j \leq a$ where s_j and t_j are terms in the congruence closure graph of ϕ. This is because equalities can only propagate "upwards" during congruence closure. Similarly, one only needs to consider the case in which s'_i and t'_i are in the congruence closure graph of ϕ. The formal correctness of the cover algorithm is presented in the full version of the paper [7].

Line 1 involves computing a congruence closed graph G that represents the equalities implied by ϕ. G is a set of congruence classes, and each congruence

$$\text{Formula } \phi \quad : \quad s_1 = F(z_1, v) \ \wedge \ s_2 = F(z_2, v) \ \wedge \ t = F(F(y_1, v), F(y_2, v))$$
$$\mathbb{C}v\colon \phi \quad : \quad z_1 = z_2 \Rightarrow s_1 = s_2 \ \wedge \bigwedge_{i,j \in \{1,2\}} y_1 = z_i \wedge y_2 = z_j \ \Rightarrow \ t = F(s_i, s_j)$$

Fig. 2. An example of cover operation for the theory of uninterpreted functions

class is a set of nodes n, where a node is either a variable y, or a F-node $F(c_1, c_2)$ for some congruence classes c_1 and c_2. Note that two nodes n_1 and n_2 in G are in the same congruence class iff ϕ implies $n_1 = n_2$. The function $\texttt{Rep}(c)$ returns a representative term for class c that does not involve any variables in V, if any such term exists; otherwise it returns $\perp$.

Line 2 calls procedure `Mark` that takes a congruence closed graph G and a set of variables V as inputs, and sets $M[n]$ to 1 for F-nodes n iff node n in G becomes undefined if variables V are removed from G. An F-node $F(c_1, c_2)$ is undefined iff classes c_1 or c_2 are undefined. A class c is undefined iff it contains all undefined nodes. The function `AllMark` takes a congruence class c as an input and returns true iff all nodes in c are marked.

Lines 5 through 8 compute $W[n_1, n_2]$, which denotes the weakest constraint not involving variables in V and which along with ϕ implies $n_1 = n_2$. $W[n_1, n_2]$ is first initialized to $\texttt{Init}(n_1, n_2)$, which returns a constraint not involving variables in V and which along with ϕ implies $n_1 = n_2$. $W[n_1, n_2]$ is then updated in a transitive closure style.

Line 4 initializes `result` to all equalities and disequalities that are implied by ϕ and that do not involve any variables in V. Lines 12 and 15 then update `result` by conjoining it with all implied equalities that are implied by ϕ and that do not involve any variable from V. Lines 11-12 can be treated as a special case of lines 13-15 when the context Z does not contain any holes (i.e., $k = 0$).

Example 1. Figure 2 shows an example of the cover operation over the theory of uninterpreted functions. For the formula ϕ in Figure 2, let n_1 be the node $F(y_1, v)$, n_2 be the node $F(y_2, v)$, and n be the node $F(n_1, n_2)$. The procedure `Mark` marks all the nodes in the congruence closed graph G, as every node depends on the variable v that needs to be eliminated. After executing lines 5 through 8, the algorithm computes $W[s_1, n_1]$, for instance, to be the constraint $z_1 = y_1$. Note, that an equality between z_1 and y_1 results in an equality between the nodes s_1 and n_1, and this is the weakest constraint to do so. Similarly, $W[s_1, n_2]$ is the constraint $z_2 = y_1$, and so on. For this example, the set N_e in Line 9 contains all the nodes in G. Consider the context $Z[n_1, n_2] = F(n_1, n_2)$. By choosing the node m_1 to be s_1 and m_2 to be s_1 in line 13, we obtain the formula $z_1 = y_1 \wedge z_2 = y_1 \Rightarrow t = F(s_1, s_1)$ in line 15, and so on. The result returned by the algorithm is shown in Figure 2.

Complexity: The complexity of the algorithm described in Figure 3 can be exponential in the size of the input formula ϕ. This is because there can be an exponential number of ways of choosing an appropriate sequence of k nodes $m_1, \ldots, m_k$ in line 13. Hence, the size of the cover can itself be exponential in

```
ComputeCover_uf(φ, V) =
1   Let G be the congruence closure of φ.
2   Mark(G,V);
3   let G' be the graph obtained from G after removing all nodes n s.t. M[n] = 1;
4   result ← all equalities and disequalities implied by G';
    // Compute W[n1, n2]
5   forall nodes n1, n2 ∈ G: W[n1, n2] ← Init(n1, n2);
6   forall nodes n ∈ G:
7      forall nodes n1, n2 ∈ G:
8         W[n1, n2] ← W[n1, n2] ∨ (W[n1, n] ∧ W[n, n2]);
    // Compute result
9   let Ne = {n | n ∈ G, M[n] = 1, CRep(n) ≠ ⊥};
10  forall nodes n ∈ Ne
11     forall nodes m ∈ G s.t. W[m, n] ≠ false:
12        result ← result ∧ (W[n, m] ⇒ CRep(n) = CRep(m));
13     forall contexts Z[n1, .., nk] s.t. n = Z[n1, .., nk], Vars(Z) ∩ V = ∅, ni ∈ Ne for 1 ≤ i ≤ k
14        forall nodes m1, .., mk ∈ G s.t. W[ni, mi] ≠ false and CRep(mi) ≠ ⊥ for 1 ≤ i ≤ k:
15           result ← result ∧ ( ( ⋀_{i=1}^{k} W[ci, di] ) ⇒ CRep(n) = Z[CRep(m1), ..., CRep(mk)] );
16  return result;

// Marks those nodes n (M[n] = 1) which become undefined when variables in V are removed
Mark(G,V) =
   forall nodes n ∈ G: M[n] ← 1;
   forall variables y ∉ V: M[y] ← 0;
   while any change
      forall nodes F(c1, c2): if ¬AllMark(c1) ∧ ¬AllMark(c2), M[F(c1, c2)] ← 0;

// Returns true if every node in the equivalence class c is marked
AllMark(c) =
   forall nodes n in class c: if M[n] = 1, return true;
   return false;

// Initial candidate for the weakest constraint that implies n1 = n2
Init(n1,n2) =
   if Class(n1) = Class(n2), return true;
   if n1 ≡ F(c1, c2) and n2 ≡ F(c'1, c'2)
      if Rep(c1) ≠ ⊥ ∧ Rep(c'1) ≠ ⊥ ∧ c2 = c'2, return Rep(c1) = Rep(c'1);
      if Rep(c2) ≠ ⊥ ∧ Rep(c'2) ≠ ⊥ ∧ c1 = c'1, return Rep(c2) = Rep(c'2);
      return Init(c1, c'1) ∧ Init(c2, c'2);
   return false;

// Find a term in the equivalence class not containing a variable in V
Rep(c) =
   if AllMark(c) return ⊥;
   if c has a variable y s.t. M[y] = 0, return y;
   let F(c1, c2) be the node s.t. M[F(c1, c2)] = 0. return F(Rep(c1), Rep(c2));

// Find a representative term for n that does not contain a variable in V
CRep(n) =
   return Rep(Class(n));
```

Fig. 3. Cover Algorithm for Theory of Uninterpreted Functions

size of the input formula ϕ. The formula ϕ in Figure 2 can be easily generalized to obtain a formula of size $O(n)$ whose cover is of size $O(2^n)$.

Special Case of Unary Uninterpreted Functions. For the special case when the formula ϕ involves only unary uninterpreted functions, the cover algorithm simply involves erasing variables in V from congruence closure of ϕ. Equivalently, the algorithm only involves Lines 1 through 4 in the `ComputeCover` procedure described in Figure 3. The complexity of the cover algorithm for unary uninterpreted functions is thus $O(n \log n)$, where n is the size of the input formula.

4 Combination Algorithm for Cover

In this section, we show how to obtain a cover algorithm for combination of two theories $T_1 \cup T_2$ from the cover algorithms for the individual theories T_1 and T_2. Our combination methodology is based on extension of Nelson-Oppen methodology for combining decision procedures for two theories. As a result, the restrictions on theories that allow for efficient combination of their decision procedures (namely, convexity, stably infiniteness, and disjointness) also transfer to the context of combining cover algorithms for those theories.

The Nelson-Oppen methodology for combining decision procedures involves sharing variable equalities $v = u$ between the formulas in the two theories. For combining cover algorithms, we also need to share variable-term equalities (i.e., equalities between variables and terms) apart from variable equalities. Furthermore, these equalities may also be conditional on any predicate. More formally, the general form of equalities that we share between the two formulas in the two theories is $\gamma \Rightarrow v = t$, where γ is a formula that does not involve any variable to be eliminated, and either v and t are both variables (in which case we refer to it as a *conditional variable equality*) or v is a variable that needs to be eliminated and t is a term that does not involve any variable to be eliminated (in which case we refer it to as a *conditional variable-term equality*). The terms t are restricted to come from a set that we refer to as set of *simple terms* (Definition 1).

We now introduce some notation that is needed to describe the cover algorithm for combination of two theories.

Definition 1 (Set of Simple Terms). *A set S is a set of simple terms for variable v with respect to a formula ϕ in theory T (denoted by $\mathtt{SST}_T(v, \phi)$), if for all conjunctions of atomic predicates γ such that $v \notin \mathtt{Vars}(\gamma)$, and all terms t that are distinct from v:*

$$\begin{array}{rcl} v \notin \mathtt{Vars}(S) & \textit{and} & \mathtt{Vars}(S) \subseteq \mathtt{Vars}(\phi) \\ (\gamma \wedge \phi \Rightarrow_T v = t) & \Rightarrow & \exists t' \in S \ s.t. (\gamma \wedge \phi \Rightarrow_T v = t') \ \wedge \ (\gamma \wedge \phi \Rightarrow_T t = t') \end{array}$$

We refer to t' as $ST(S, \gamma, t)$.

The theories of linear arithmetic and uninterpreted functions admit a finite set of simple terms for their formulas. The following theorems describe how to compute a set of simple terms for a formula in the corresponding theory.

Theorem 3 (Set of Simple Terms for Linear Arithmetic). *Let ϕ be the formula $\bigwedge_{i=1}^{n} v \leq a_i \wedge \bigwedge_{i=1}^{m} v \geq b_i \wedge \bigwedge_{i=1}^{n'} v < a'_i \wedge \bigwedge_{i=1}^{m'} v > b'_i \wedge \phi'$. where $v \notin \mathtt{Vars}(\phi'), v \notin \mathtt{Vars}(a_i)$ and $v \notin \mathtt{Vars}(b_i)$). Then, $\{a_i\}_{i=1}^{n}$ is $\mathtt{SST}_{\ell\mathtt{a}}(v, \phi)$.*

Theorem 4 (Set of Simple Terms for Uninterpreted Functions). *Let ϕ be a formula over the theory of uninterpreted functions. Let G be the congruence closure of ϕ. Let t be any term in the congruence class of v in G that does not involve v (if any such term exists). Then, the singleton set containing t is $\mathtt{SST}_{\mathtt{uf}}(v, \phi)$ (if any such term exists). If no such term exists then, $\mathtt{SST}_{\mathtt{uf}}(v, \phi) = \emptyset$.*

The proofs of Theorem 3 and Theorem 4 are given in the full version of the paper [7].

We use the notation $WC_T(\phi, \delta, V)$ to denote $\neg \mathbb{C}_T V : \phi \wedge \neg\delta$. Intuitively, $WC_T(\phi, \delta, V)$ denotes the weakest constraint that together with ϕ implies δ and that does not involve any variable from set V.

The following property is useful in describing $\mathtt{Cover}_{T_1 \cup T_2}(\phi, V)$ in terms of $\mathtt{Cover}_{T_1}$ and $\mathtt{Cover}_{T_2}$.

Property 4. Let ϕ and ϕ' be quantifier-free formulas in theory T such that

$$\begin{aligned} \phi &\Rightarrow_T \phi' \\ V \cap \mathtt{Vars}(\phi') &= \emptyset \\ (V \cap \mathtt{Vars}(\gamma) = \emptyset \ \wedge\ \phi \Rightarrow_T \gamma) &\Rightarrow (\phi' \Rightarrow_T \gamma), \text{ for all quantifier-free formulas } \gamma \end{aligned}$$

Then, $\phi' \equiv \mathtt{Cover}_T(\phi, V)$.

We use the notation $\mathtt{Num}(T)$ for any theory T to denote the maximum number of variables that may occur in any atomic predicate in theory T. For example, $\mathtt{Num}(T) = 2$ for difference logic (theory of linear arithmetic with only difference constraints) as well as for theory of unary uninterpreted functions.

The procedure $\mathtt{ComputeCover}_{T_1 \cup T_2}$ in Figure 4 takes as input a formula ϕ and a set of variables V to be eliminated and computes $\mathbb{C}_{T_1 \cup T_2} V : \phi$ using the cover algorithms for theories T_1 and T_2. Line 1 performs purification of ϕ, which involves decomposing ϕ (which is a conjunction of atomic predicates in the combined theory $T_1 \cup T_2$) into conjunctions of atomic predicates that are either in theory T_1 or in T_2 by introducing a fresh variable for each *alien* term in ϕ. The set of all such fresh variables is referred to as U, while V' denotes the set of all variables that we need to eliminate from $\phi_1 \wedge \phi_2$. Lines 4 to 11 repeatedly exchange conditional variable equalities and conditional variable-term equalities between ϕ_1 and ϕ_2. Lines 13 and 14 call the procedure $\mathtt{ComputeSimpleCover}_{\mathtt{T}}$, which takes as inputs a set of variables V, a formula ϕ in theory T, and a formula F of the form $\bigwedge \gamma_i \Rightarrow_{T'} v_i = t_i$ (where $v_i \in V$ and T' is any theory) such that $V \cap (\mathtt{Vars}(\gamma_i) \cup \mathtt{Vars}(t_i)) = \emptyset$, and computes $\mathbb{C}_{T \cup T'} V : \phi \wedge F$.

The proof of correctness (including termination) of the combination algorithm in Figure 4 is non-trivial and is given in the full version of the paper [7]. We give a brief sketch of the proof here. Let γ_i's be some atomic predicates that do not

```
ComputeCover_{T1∪T2}(V, φ) =
1   φ1, φ2 = Purify(φ); let U be the variables introduced during Purify(φ);
2   let V' = V ∪ U;
3   F1 ← true; F2 ← true;
4   repeat until no change:
5      for j = 1, 2:
6         let ⋀_{i=1}^{n} γi ⇒ vi = ui be some conditional variable equalities implied by F_{3−j};
7         let ⋀_{i=1}^{m} δi ⇒ wi = ti be some conditional variable-term equalities implied by F_{3−j};
8         let ψ = ⋀_{i=1}^{n} γi ∧ ⋀_{i=1}^{m} δi; let E = ⋀_{i=1}^{k} vi = ui; let W = V' − {wi | 1 ≤ i ≤ m};
9         let Sv ≡ SST_{Tj}(v, C_{Tj} W − {v} : φj ∧ E) for any variable v;
10        Fj ← Fj ∧ ⋀_{v1,v2∈V'} ψ ∧ WC_{Tj}(φj ∧ E, v1 = v2, W)[ti/wi] ⇒ v1 = v2
11                ∧ ⋀_{v∈V',t∈Sv} ψ ∧ WC_{Tj}(φj ∧ E, v = t, W)[ti/wi] ⇒ v = t
12   let F1' be the conjunction of all implied variable-term equalities γi ⇒ vi = ti
                                 implied by Fj s.t. Vars(γi) ∩ V' = ∅ (for j = 1, 2);
13   let α1 = ComputeSimpleCover_{T1}(V', φ1, F2');
14   let α2 = ComputeSimpleCover_{T2}(V', φ2, F1');
15   return α1 ∧ α2;

ComputeSimpleCover_T(V, φ, F) =
   result ← ComputeCover(V, φ);
   forall collections ⋀_{i=1}^{m} γi ⇒ wi = ti of conditional variable-term equalities implied
                                 by F s.t. m ≤ Num(T) and wi are all distinct variables:
      let γ = ⋀_{i=1}^{n} γi; let W = V − {wi | 1 ≤ i ≤ m};
      result ← result ∧ (γ ⇒ ComputeCover(W, φ)[ti/wi]);
   return result;
```

Fig. 4. Cover algorithm for combination of two theories $T_1 \cup T_2$

involve variables in V and furthermore $\phi \Rightarrow \gamma_1 \vee \ldots \vee \gamma_k$. We show that the formula $\texttt{ComputeCover}_{T_1 \cup T_2}(V, \phi) \wedge \neg\gamma_1 \wedge \ldots \wedge \neg\gamma_k$ is unsatisfiable by simulating the decision procedure for theory $T_1 \cup T_2$ based on Nelson-Oppen's combination methodology (with the knowledge of the Nelson-Oppen proof of unsatisfiability of the formula $\phi \wedge \neg\gamma_1 \wedge \ldots \wedge \neg\gamma_k$).

The complexity of the cover algorithm for combination of two theories is an exponential (in size of the input formula ϕ and cardinality of its set of simple terms) factor of the complexity of the cover algorithms for individual theories. For combination of difference logic (theory of linear arithmetic with only difference constraints) and unary uninterpreted functions, which is a useful combination that occurs in practice, the cover algorithm can be simplified and it runs in time polynomial in size of the input formula ϕ.

We now present some examples of computation of cover for the combined theory of linear arithmetic ($\ell\texttt{a}$) and uninterpreted functions ($\texttt{uf}$). Example 2 demonstrates the importance of sharing variable-term equalities, while Example 3 demonstrates the importance of sharing conditional equalities.

Example 2. Compute $\mathbb{C}_{\ell\texttt{a}\cup\texttt{uf}}\{v_1, v_2\} : \phi$, where ϕ is $(a \leq v_1 + 1 \wedge v_1 \leq a - 1 \wedge v_2 \leq b \wedge v_1 = F(v_3) \wedge v_2 = F(F(v_3)))$, for some uninterpreted function F.

We first decompose ϕ into pure formulas ϕ_1 and ϕ_2:

$$\phi_1 = (a \leq v_1 + 1 \ \wedge \ v_1 \leq a - 1 \ \wedge \ v_2 \leq b)$$
$$\phi_2 = (v_1 = F(v_3) \ \wedge \ v_2 = F(F(v_3)))$$

We then share variable-term equalities between ϕ_1 and ϕ_2 as follows:

$$\phi_1 \xrightarrow{v_1 = a-1} \phi_2 \xrightarrow{v_2 = F(a-1)} \phi_1$$

We then compute $\mathbb{C}_{\ell_{\mathtt{a}}}\{v_1, v_2\} : \phi_1 \wedge v_2 = F(a-1)$ to obtain the result $F(a-1) \leq b$. Note that the cover algorithm for linear arithmetic does not need to understand the term $F(a-1)$ and can just treat it as some fresh variable.

Example 3. Compute $\mathbb{C}_{\ell_{\mathtt{a}\cup\mathtt{uf}}} x : \phi$, where ϕ is $(y = \mathtt{Mem}(a + x) - \mathtt{Mem}(b + x))$ for some uninterpreted function `Mem`.
Purifying ϕ, we obtain ϕ_1 and ϕ_2 by introducing new variables u_1, u_2, u_3, u_4.

$$\phi_1 = (y = u_1 - u_2 \ \wedge \ u_3 = a + x \ \wedge \ u_4 = b + x)$$
$$\phi_2 = (u_1 = \mathtt{Mem}(u_3) \ \wedge \ u_2 = \mathtt{Mem}(u_4))$$

We then share conditional equalities between ϕ_1 and ϕ_2 as follows:

$$\phi_1 \xrightarrow{a=b \Rightarrow u_3 = u_4} \phi_2 \xrightarrow{a=b \Rightarrow u_1 = u_2} \phi_1$$

We then compute $\mathbb{C}_{\ell_{\mathtt{a}}}\{x, u_1, u_2, u_3, u_4\} : \phi_1 \wedge a = b \Rightarrow u_1 = u_2$ to obtain the result $a = b \Rightarrow y = 0$.

5 Related Work

5.1 Discovering Invariants over Combination of Linear Arithmetic and Uninterpreted Functions

There has been some work on generating conjunctive invariants that involve combination of linear arithmetic and uninterpreted functions. [10] discovers invariants over a given set of terms, while generates invariants over programmer specified templates [2]. Our approach (extended with a suitable widening operation) can be used to discover (possibly disjunctive) invariants over the combination of theories without the need to provide any terms/templates.

5.2 Abduction

An important key idea used in the cover algorithms described in this paper is that of *abduction*. Abduction is reasoning from an observation to its best explanation. More formally, an abductive explanation of observation ψ given environment E in language L is a formula $\psi' = \mathtt{Abduct}(E, \psi, L)$ such that $\psi' \wedge E \Rightarrow \psi$, ψ' is in L, and ψ' is the weakest such formula. The notion of abduction is widely used in the artificial intelligence community [17] and in the logic programming community [12].

The array $W[n_1, n_2]$ (computed in lines 5 through 8 in Figure 3) used in the algorithm for computing cover of a formula ϕ with respect to variables V in the theory of uninterpreted functions is essentially $\mathtt{Abduct}(\phi, n_1{=}n_2, L)$, where L is the language of formulas over variables other than V. Similarly, the formula $WC(\phi, \delta, V)$ used in the combination cover algorithm is essentially $\mathtt{Abduct}(\delta, \phi, L)$.

5.3 Predicate Cover

The notion of cover discussed in this paper is similar to the *predicate cover* [13] operation used in predicate abstraction algorithms. For a formula ϕ, predicate cover is the weakest Boolean formula over a set of given predicates that implies ϕ. In contrast, the cover of ϕ is defined over a much richer language — the set of all quantifier-free formulas.

6 Conclusion and Future Work

This paper defines cover as the most precise quantifier-free over-approximation to existential quantifier elimination, and describes algorithms to compute the cover of formulas in the theories of uninterpreted functions and linear arithmetic. In addition, this paper provides a combination algorithm to combine the individual cover algorithms for these theories. This paper also describes how the cover operation can be used in program analysis and verification techniques that otherwise require existential quantifier elimination.

We hope to extend this study in future work. We are currently exploring the implementation of the symbolic model checking algorithm described in this paper. Also, the notion of cover can be parameterized by types of formulas that one is interested in. Instead of generating the most precise quantifier-free formula, one may be interested in formulas that are conjunctions of, say, atomic predicates, or at most k disjunctions of atomic predicates, or implications of the form $\phi_1 \Rightarrow \phi_2$, where ϕ_1 and ϕ_2 are conjunctions of atomic predicates in variables V_1 and V_2 respectively. The latter may be useful in computing procedure summaries, where V_1 and V_2 denote the set of input and output variables respectively.

References

1. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: 29th Annual Symposium on POPL, pp. 1–3 (2002)
2. Beyer, D., Henzinger, T., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007)
3. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. Transactions on Software Engg. 30(6), 388–402 (2004)
4. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
5. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: 5th ACM Symposium on POPL, pp. 84–96 (1978)
6. Gulavani, B., Rajamani, S.: Counterexample driven refinement for abstract interpretation. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, Springer, Heidelberg (2006)
7. Gulwani, S., Musuvathi, M.: Cover algorithms and their combination. Technical Report MSR-TR-2006-09, Microsoft Research (January 2006)

8. Gulwani, S., Necula, G.C.: Global value numbering using random interpretation. In: 31st Annual ACM Symposium on POPL (January 2004)
9. Gulwani, S., Necula, G.C.: A polynomial-time algorithm for global value numbering. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 212–227. Springer, Heidelberg (2004)
10. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: PLDI, pp. 376–386 (2006)
11. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: 31st Annual ACM Symposium on POPL, pp. 232–244 (2004)
12. Kakas, A.C., Kowalski, R.A., Toni, F.: Abductive logic programming. Journal of Logic and Computation 2(6), 719–770 (1992)
13. Lahiri, S.K., Ball, T., Cook, B.: Predicate Abstraction via Symbolic Decision Procedures. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 24–38. Springer, Heidelberg (2005)
14. McMillan, K.: Symbolic model checking: an approach to the state explosion problem. PhD thesis, Carnegie Mellon University (1992)
15. McMillan, K.: An Interpolating Theorem Prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004)
16. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM TOPLAS 1(2), 245–257 (1979)
17. Paul, G.: Ai approaches to abduction, 35–98 (2000)