Heuristics for Efficient Manipulation of Composite Constraints

Tuba Yavuz-Kahveci and Tevfik Bultan

Department of Computer Science, University of California,
Santa Barbara, CA 93106 USA
{tuba, bultan}@cs.ucsb.edu

Abstract. Composite Symbolic Library is a symbolic manipulator for model checking systems with heterogeneous data types. Our current implementation uses two basic symbolic representations: BDDs for boolean and enumerated variables, and polyhedra for (unbounded) integers. These basic representations are imported to the Composite Symbolic Library using a common interface and are combined using a disjunctive composite representation. In this paper, we present several heuristics for efficient manipulation of this composite representation. Our heuristics make use of the following observations: 1) efficient operations on BDDs can be used to mask expensive operations on polyhedra, 2) our disjunctive representation can be exploited by computing pre and post-conditions and subset checks incrementally, and 3) size of a composite representation can be minimized by iteratively merging matching constraints and removing redundant ones. We present experimental results that illustrate efficiency of our algorithms.

1 Introduction

The introduction of compact and efficient symbolic representations has enabled automated verification of large hardware and software systems by overcoming the state-space explosion problem [BCM+90,McM93,CAB+98]. Symbolic representations are efficient alternatives to explicit state exploration, since they provide a compact representation of the state space. Properties of a system can be verified by manipulating the symbolic representations that represent its transition relation and states.

Binary Decision Diagrams (BDDs) [Bry86] (for representing boolean constraints) and polyhedral representation [Hal93] (for linear arithmetic constraints) are two examples for such symbolic representations. BDDs have been successfully used in verification of finite-state systems which could not be verified explicitly due to size of the state space [BCM+90,McM93,CAB+98]. Linear arithmetic constraint representations have been used in verification of real-time systems, and infinite-state systems [AHH96,BGP99,HRP94] which are not possible to verify using explicit representations.

One problem with these symbolic representations is that they are specialized for certain domains; i.e., BDDs are specialized for encoding boolean variables and polyhedral representation is specialized for representing states of integer and real variables as linear arithmetic constraints. As a result, BDDs are restricted to finite domains and polyhedral representation becomes inefficient when it is used for a large set of boolean variables. However, it is very likely that a large software specification would use more than one data type. Motivated with this observation, we have designed and implemented the Composite Symbolic Library [YKTB01] that combines several symbolic representations. Our current implementation uses BDDs for boolean and enumerated variables and polyhedra for (unbounded) integers. However, Composite Symbolic Library can be easily extended with new symbolic representations since we have adopted an object-oriented approach in its design.

In this paper, we present several heuristics for efficient manipulation of this composite representation. Our heuristics make use of the following observations: 1) efficient operations on BDDs can be used to mask expensive operations on polyhedra, 2) our disjunctive representation can be exploited by computing pre and post-conditions and subset checks incrementally, and 3) size of a composite representation can be minimized by iteratively merging matching constraints and removing redundant ones. We have implemented the above ideas and experimented with a large set of examples. Experimental results indicate effectiveness of our heuristics.

Techniques that are similar to our heuristics have been used in the literature. In [DP01], local subsumption test is used during the fixpoint computations to remove the redundant constrained facts. This is similar to our approach for preventing the increase in the size of the disjunctive composite representation during fixpoint computation by removing redundant disjuncts. However, we use full subsumption test. Local subsumption test can also be used as a heuristic to test the convergence of fixpoint computations [DP01]. However, there can be cases where fixpoint computation that uses the local subsumption test does not converge whereas the fixpoint computation that uses the full subsumption test converges.

Hytech, a tool for verification of hybrid systems, simplifies formulas using rewrite rules [AHH96]. The approach used in [AHH96] is for simplification of linear arithmetic formulas on real variables. Our work is different in two respects: 1) We use linear arithmetic formulas on integer variables. 2) Our heuristics are not for simplification of linear arithmetic formulas, this is handled by the constraint manipulator we use [Ome]. Rather, our heuristics are for simplification of composite formulas which contain a mixture of boolean and integer variables. In Hytech tool boolean or enumerated variables (for example control states) are eliminated by partitioning the state space [AHH96].

In [Sri93], a linear partitioning algorithm for convex polyhedra is used to efficiently test if a single convex polyhedron is subsumed by a union of convex polyhedra. This approach is analogous to our subset check heuristic where the union of convex polyhedra corresponds to our disjunctive composite representation and the single convex polyhedron corresponds to a single disjunct of composite representation.

The rest of the paper is organized as follows. In Section 2 and Section 3, we explain Composite Symbolic Library and our model checker that uses the composite symbolic representation, respectively. In Section 4, we present heuristics that masks expensive polyhedral operations with BDD operations. We introduce an efficient subset check algorithm for composite symbolic representation in Section 5. In Section 6, we propose several algorithms with different aggressiveness levels for minimization of composite symbolic representation. We explain how we can improve a fixpoint algorithm in Section 7. In Section 8, we present the experimental results of the heuristics that we propose in Sections 3-7. Finally, in Section 9, we give our conclusions.

2 Composite Symbolic Library

Composite Symbolic Library is a symbolic manipulator for model checking systems with heterogeneous data types. In Composite Symbolic Library different symbolic representations are combined using the *composite model checking* approach [BGL00]. Our current implementation of the Composite Symbolic Library uses two symbolic representations: BDDs for boolean logic formulas and polyhedral representation for Presburger arithmetic formulas. We call these representations *basic symbolic representations*.

Each variable type in the system to be verified is assigned to the most efficient basic symbolic representation for that variable type. Boolean and enumerated variables are mapped to BDD representation, and integers are mapped to arithmetic constraint representation. We encode the sets of system states and the transition relation of the system in Disjunctive Normal Form (DNF), as a disjunction of conjunctions of basic symbolic representations (e.g., a disjunct consists of conjunction of a boolean formula stored as a BDD representing the states of boolean and enumerated variables, and a Presburger arithmetic constraint representing the states of integer variables). We call this DNF representation a *composite symbolic representation* since it combines different basic symbolic representations.

We used an object-oriented design for the Composite Symbolic Library. An abstract class called `Symbolic` serves as an interface to all symbolic representations in the Composite Symbolic Library including the composite representation. A class called `BoolSym` serves as a wrapper for the BDD library CUDD [CUD] and derived from abstract class `Symbolic`. Similarly, `IntSym` class is also derived from abstract class `Symbolic` and serves as a wrapper for the Omega Library [Ome]. A class called `CompSym` is the class for composite representations. It is derived from `Symbolic` and uses `IntSym` and `BoolSym` (through the `Symbolic` interface) to manipulate composite representations.

The object-oriented design for the Composite Symbolic Library has several advantages: 1) The manipulation of the composite representations is independent of the manipulation of the basic symbolic representations and number of basic symbolic representations. `CompSym` accesses basic symbolic representations using only the `Symbolic` interface and it uses the number of basic symbolic representations as a parameter. 2) It is easy to replace the Omega Library and CUDD Library with other symbolic manipulators as long as one writes a wrapper around the new symbolic manipulator which conforms to the `Symbolic` interface. 3) Verification is polymorphic. Since the verification procedures use the `Symbolic` interface they work both for composite symbolic representations and basic symbolic representations. I.e., based on the input specification our current implementation can be used as a BDD-based model checker, a polyhedra-based model checker or a composite model checker.

To analyze a system using Composite Symbolic Library, one has to specify its initial condition, transition relation, and state space using a set of *composite formulas*. A composite formula is obtained by combining boolean and integer formulas with logical connectives. A boolean formula consists of boolean variables or constants joined with logical connectives. An integer formula consists of integer variables or constants joined with arithmetic operators $+$ and $-$, arithmetic predicates $<, >, =, \leq, \geq$, logical connectives, and quantifiers $\exists$ and $\forall$. Since symbolic representations in the Composite Symbolic Library currently support only Presburger arithmetic formulas, we restrict arithmetic operators to $+$ and $-$. However, we allow multiplication with a constant and quantification.

In `CompSym`, a composite formula, P, is represented in DNF as

$$P = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{t} p_{ij}$$

where p_{ij} denotes the formula of basic symbolic representation type j in the ith disjunct, and n and t denote the number of disjuncts and the number of basic symbolic representation types, respectively. We call each disjunct $\wedge_{j=1}^{t} p_{ij}$ a *composite atom*. Each composite atom is implemented as an instance of a class called `compAtom`. The `compAtom` class has a field called `representation`. `representation` field is an array of class `Symbolic` and the size of the array is the number of basic symbolic representations. Semantically, each `compAtom` object represents a conjunction of formulas each of which is either a boolean or an integer formula (which are represented as `BoolSym` and `IntSym` objects, respectively). A composite formula stored in a `CompSym` object is implemented as a list of `compAtom` objects, which corresponds to the disjunction in the DNF above.

The `CompSym` class includes methods (such as intersection, union, complement, satisfiability check, equivalence check, etc.) which manipulate composite representations in the above form. These methods in turn call the related method of basic symbolic methods (which are implemented in `IntSym` and `BoolSym` classes). Note that all these operations can be effectively computed both for boolean logic formulas and Presburger arithmetic formulas.

3 Symbolic Verifier

Given a set of states S and a transition relation R, pre-condition $\textsc{pre}(S, R)$ are all the states that can reach a state in S with a single transition in R

(i.e., the set of predecessors of all the states in S). Post-condition $\textsc{post}(S,R)$ is defined similarly. Given a set S and a transition relation R both represented using composite symbolic representation as $S \equiv \bigvee_{i=1}^{n_S} \bigwedge_{j=1}^{t} s_{ij}$ and $R \equiv \bigvee_{i=1}^{n_R} \bigwedge_{j=1}^{t} r_{ij}$ the pre-condition can be computed as

$$\textsc{pre}(S,R) \equiv \bigvee_{i=1}^{n_R} \bigvee_{k=1}^{n_S} \bigwedge_{j=1}^{t} \textsc{pre}(s_{kj}, r_{ij})$$

The above property holds because the existential variable elimination in the $\textsc{pre}(S,R)$ computation distributes over the disjunctions, and due to the partitioning of the variables based on the basic symbolic types, the existential variable elimination also distributes over the conjunction above.

Our symbolic model checker computes fixpoints for CTL temporal operators using the function $\textsc{pre}(S,R)$ and fixpoint computations [BYK01]. In the experiments reported in this paper we use the basis $\{\mathrm{EX}, \mathrm{EG}, \mathrm{EU}\}$ for CTL formulas [CGP99].

4 Masking Integer Operations

Composite Symbolic Library currently supports two basic symbolic representations: BDDs to represent boolean and enumerated variables and polyhedral representation of linear arithmetic constraints to represent integer variables. Existential variable elimination for linear integer arithmetic constraints is NP-complete and it is used in satisfiability check and pre and post condition computations. However, since BDD representation is canonical, satisfiability check for BDDs can be performed in constant time by comparing the root node to the unique BDD that corresponds to *false*. This discrepancy in the performances of BDDs and polyhedral representation in checking satisfiability can be exploited to speed up pre and post condition computation on the composite symbolic representation.

A composite atom $s = b \wedge i$, where b and i denote the BDD part and polyhedral part, respectively, is satisfiable iff both b and i are satisfiable. Since satisfiability check for polyhedral representation is expensive, by checking the satisfiability of the BDD part first we can avoid checking satisfiability for the polyhedral part whenever the BDD part is not satisfiable. If we find out that the BDD part is not satisfiable we can conclude that the composite atom is not satisfiable.

Given two composite formulas $S = \bigvee_{j=1}^{n_S} b_{sj} \wedge i_{sj}$ and $R = \bigvee_{k=1}^{n_R} b_{rk} \wedge i_{rk}$, where S represents a set and R represents the transition relation, b_{sj} and b_{rk} correspond to boolean formulas, and i_{sj} and i_{rk} correspond to integer formulas, pre-condition of S with respect to R can be written as

$$\textsc{pre}(S,R) = \bigvee_{j=1}^{n_S} \bigvee_{k=1}^{n_R} \textsc{pre}(b_{sj}, b_{rk}) \wedge \textsc{pre}(i_{sj}, i_{rk})$$

Instead of computing $\textsc{pre}(b_{sj}, b_{rk})$ and $\textsc{pre}(i_{sj}, i_{rk})$ and then taking the intersection of the two, we can first compute $\textsc{pre}(b_{sj}, b_{rk})$ and then check it for satisfiability. Since $\mathrm{PRE}(b_{sj}, b_{rk})$ is a boolean formula and represented by BDDs, checking satisfiability of $\mathrm{PRE}(b_{sj}, b_{rk})$ is cheaper than checking satisfiability of $\mathrm{PRE}(i_{sj}, i_{rk})$, which is represented by polyhedra. We should compute $\mathrm{PRE}(i_{sj}, i_{rk})$ that involves manipulation of polyhedral representation only if $\mathrm{PRE}(b_{sj}, b_{rk})$ is satisfiable. If it is not satisfiable then we will not compute $\mathrm{PRE}(i_{sj}, i_{rk})$ since we can deduce that $\mathrm{PRE}(b_{sj}, b_{rk}) \wedge \mathrm{PRE}(i_{sj}, i_{rk})$ evaluates to *false*. As a result expensive integer manipulation is masked by cheaper boolean manipulation.

5 Subset Check

A composite formula $A = \bigvee_{i=1}^{n} a_i$ is subset of a composite formula $B = \bigvee_{k=1}^{m} b_k$ iff $\forall i$ s.t. $1 \leq i \leq n$, $a_i \subseteq B$. So the most straightforward way of checking subset relation between composite formulas A and B is to iterate through the composite atoms in A and check subset relation between each composite atom a_i in A and B. If there exists a composite atom a_i in A such that a_i is not subset of B we can conclude that A is not subset of B. On the other hand, if there exists no such composite atom in A then we can conclude that A is subset of B.

Figure 1(a) shows the algorithm for checking subset relation between two composite formulas A and B. For each composite atom a_i in A, first the algorithm checks if a_i is subset of any composite atom in B (lines 4-8). A composite atom $a = \bigwedge_{i=1}^{t} a_i$ is subset of composite atom $b = \bigwedge_{i=1}^{t} b_i$ iff $\forall i$ s.t. $1 \leq i \leq t$, $a_i \subseteq b_i$, where t is the number of basic types. If there exists no composite atom b in B such that a is subset of b then this does not mean that a is not subset of B. Next, the algorithm computes $a \cap \neg B$ and assigns the result to composite formula C. If C is satisfiable then it means a is not subset of B and the algorithm exits by returning false (lines 9-12). Otherwise the algorithm continues until either it finds out that there exists a composite atom a that is not subset of B or it has checked all the composite atoms in A in which case it returns true (line 13).

Time complexity of the algorithm in Figure 1(a) is

$$O(n_A \times n_B \times \textstyle\sum_{j=1}^{t} T^{j}_{IsSubset} + n_A \times (\sum_{i=1}^{n_B} \sum_{j=1}^{t} T^{j}_{Complement} + \\ t^{n_B} \times \sum_{j=1, 1 \leq t_j \leq t}^{n_B} T^{t_j}_{Intersection} + n_B \times \sum_{j=1}^{t} T^{j}_{IsEmpty})).$$

where n_A, n_B, t, and T^{i}_{Op} are number of composite atoms in A, number of composite atoms in B, number of basic symbolic representations and time complexity of operation Op for ith basic symbolic representation, respectively. Expensive part of the algorithm is computing the complement of composite formula B (line 10) since time complexity of complement operation on a composite formula B is exponential in the number of composite atoms in B. For the subset check algorithm, computing complement of B means that all the composite atoms in B are taken into consideration to decide if a composite atom a is subset of B. However, deciding if a composite atom a is subset of a composite formula B does not always require to consider all the composite atoms in B. For instance, let composite atom a and composite formula B be,

$$a = (x \wedge y \wedge (z > 0 \vee z < -4)),\ B = (x \wedge z \geq 0) \vee (x \wedge z \leq -1) \vee (\neg x \wedge z < 0)$$

```
IsSubset(A, B): boolean
    found: boolean
    A, B, C: composite formula
    for each composite atom a in A do
        found ← false
        for each composite atom b in B do
            if a ⊆ b then
                found ← true
                break;
        if ¬found then
            C ← a ∩ ¬B
            if C ≠ ∅ then
                return false;
    return true;
```

(a)

```
IsSubset(A, B): boolean
    A, B, U, t: composite formula
    for each composite atom a in A do
        U ← a
        for each composite atom b in B do
            for each composite atom u in U do
                t ← b ∩ u
                if t ≠ ∅ then
                    t ← u ∩ ¬t
                    remove u from U
                    if t ≠ ∅ then
                        U ← U ∪ t
            if size(U) = 0 then
                break;
        if size(U) ≠ 0 then
            return false;
    return true;
```

(b)

Fig. 1. Subset check algorithms performing negation at (a) composite formula level and at (b) composite atom level

where x and y are boolean variables and z is an integer variable. Since each composite atom in a composite formula corresponds to a disjunct of the composite formula, B has three composite atoms b_1, b_2, and b_3 that correspond to $(x \wedge z \geq 0)$, $(x \wedge z \leq -1)$, and $(\neg x \wedge z < 0)$, respectively. In order to decide if a is subset of B we do not need to consider all the composite atoms in B. For this example, it is sufficient to compare a against b_1 and b_2 (note that a is subset of $b_1 \vee b_2$) only to conclude that a is subset of B. However, the algorithm given in Figure 1(a) will process b_1, b_2, and b_3 by computing complement of B.

In the light of this observation we propose a more efficient solution to subset check problem for composite formulas. Given two composite formulas A and B, for each composite atom a in A, our solution iteratively computes uncovered subset of a, U, that is not covered by the composite atoms in B that have been examined so far. U is initialized to a and for each k s.t. $1 \leq k \leq n_B$, U is updated as $U \cap \neg b_k$. After U is updated using b_k it is checked for emptiness. If it becomes empty then the algorithm skips checking the remaining composite atoms in B and concludes that a is subset of B. Otherwise, it continues with b_{k+1}. After checking all composite atoms in B if U is not empty then the algorithm concludes that a is not subset of B. The algorithm is given in Figure 1(b). Note that in this algorithm there is no complement operation on the composite formula level. Instead complement is computed for composite atoms in B as needed. Time complexity of subset check algorithm in Figure 1(b) is $O(n_A \times (t-1)^{n_B} \times \sum_{i=1}^{t}(T^i_{Satisfiability} + T^i_{Intersection})$. Even though this algorithm has also an exponential worst case time complexity, in the average case we expect it to perform better than the algorithm in Figure 1(a).

6 Simplification Algorithm

The number of composite atoms in a composite formula which results from the intersection operation is linear in the product of the number of composite atoms of the input composite formulas. The number of composite atoms in a composite formula which results from the negation operation is exponential in the number of composite atoms of the input composite formula. Most of the time these resulting composite formulas are not minimal in terms of the number of composite atoms they have. For instance, composite formula A that represents the formula $(x \wedge y = z + 1) \vee (t \wedge y = z + 1) \vee ((x \vee t) \wedge y > z)$, where x and t are boolean variables and y and z are integer variables, has three composite atoms that correspond to the three disjuncts $(x \wedge y = z+1)$, $(t \wedge y = z+1)$, and $((x \vee t) \wedge y > z)$. However, A can be equivalently composed of a single composite atom that represents the formula $((x \vee t) \wedge y > z)$. Since time complexity of manipulating composite formulas is dependent on the number of composite atoms we need to reduce the number of composite atoms in a composite formula as much as possible to make the verification feasible in terms of both time and memory. We present a simplification algorithm that can be tuned for 4 different degrees of aggressiveness. First we would like to present the most aggressive version of the algorithm and then explain how aggressiveness can be traded for efficiency in a reasonable way.

A composite formula having two composite atoms, a and b, can be simplified and represented by a single composite atom c if one of the following holds: 1) a is subset of b. In this case $c = b$. 2) a is superset of b. In this case $c = a$. 3) There exists a basic type j s.t. for all basic types i s.t. $i \neq j$, $a_i = b_i$. In this case for all i s.t. $i \neq j$, $c_i = a_i$ and $c_j = a_j \vee b_j$.

Figure 2 shows the simplification algorithm. The algorithm takes each pair of composite atoms a and b in a composite formula and checks if union of the formula represented by a and b can be represented by a single composite atom r based on the above rules. The algorithm stops when there exists no two composite atoms in the composite formula that can be replaced by a single composite atom. The steps that make the algorithm aggressive are at lines 22 and 23. At line 22 the composite atom r that can replace a and b is inserted to the head of the list of composite atoms and a is set to the head of the list at line 23. This ensures that composite atom r can be compared against all other composite atoms in the list.

We can loosen the post condition of the simplification algorithm and make the algorithm less aggressive and more efficient in three ways. The first way exploits the fact that equivalence check on BDDs is cheaper than equivalence check on polyhedral representation. At line 14 of the *Simplification* algorithm in Figure 2, a_t and b_t, where t is a basic type, are checked for equivalence. Instead of checking equivalence of a_t and b_t for each basic type t, we can check it for only boolean type. This makes the *Simplification* algorithm less expensive and less aggressive meaning that given two composite atoms a and b, the *Simplification* algorithm will be able to combine a and b into a single composite atom r if a

```
Simplify(composite formula A)
1   a, b, r: composite atom
2   success: boolean
3   list_A: list of composite atoms
4   let list_A be the list of composite atoms in A
5   a ← head(list_A)
6   while a ≠ NULL do
7       b ← next(a)
8       while b ≠ NULL do
9           if a ⊆ b then remove a from list_A; break
10          else if b ⊆ a then temp ← b ; b ← next(b) ; remove temp from list_A
11          else
12              success ← false
13              for each basic type t do
14                  if a_t ≠ b_t then
15                      if ¬success then index ← t ; success ← true
16                      else success ← false ; break
17              if success then
18                  remove a and b from list_A
19                  for each basic type t do
20                      if index = t then r_t ← a_t ∨ b_t
21                      else r_t ← a_t
22                  insert r to head of list_A
23                  a ← head(list_A) ; break
24              else b ← next(b)
25      a ← next(a)
```

Fig. 2. An algorithm for simplifying a given composite formula

(b) is subset of b (a) or $a_{boolean}$ is equal to $b_{boolean}$. If $a_{boolean}$ is not equal to $b_{boolean}$ the algorithm will conclude that a and b cannot be combined.

The second way is to avoid subset check. At lines 9 and 10 *Simplification* algorithm in Figure 2 checks if a is subset of b or vice versa, respectively. By eliminating subset check between a and b the algorithm will combine two composite formula a and b only if $a_{boolean}$ is equal to $b_{boolean}$.

The third way is to consider only a subset of composite atom pairs in a given composite formula. At line 22 of the *Simplify* algorithm in Figure 2, composite atom r, which is combination of composite atoms a and b, is inserted at the head of $list_A$ and a is set to the head of $list_A$ at line 23. These two steps ensure that composite formula A is minimal by forcing analysis of every pair of composite atoms in A. However, if r is inserted at the end of $list_A$ without making a point to head of $list_A$ r is not compared with other composite atoms and the resulting composite formula A is no longer minimal (i.e., there may be pairs of composite atoms in A which can be merged).

By using the three ways of aggressiveness reduction we obtain 4 different versions of *Simplification* algorithm which we represent by $S1$, $S2$, $S3$, and $S4$ in increasing aggressiveness order. Let n denote the number of composite atoms in the input composite formula:

```
ComputeEF(composite formula P, composite formula R): composite formula
1  S_new ← P
2  do
3      S_old ← S_new
4      S_new ← PRE(S_old, R) ∪ S_old
5      S_new ← Simplify(S_new)
6  while ¬isSubset(S_new, S_old)
7  return S_new
```

Fig. 3. An algorithm for computing the least fixed point for EF

- $S1$: Number of executions of the inner while loop is $O(n^2)$ and checks equivalence relation only on boolean type and eliminates subset check.
- $S2$: Number of executions of the inner while loop is $O(n^3)$ and checks equivalence relation only on boolean type and eliminates subset check.
- $S3$: Number of executions of the inner while loop is $O(n^3)$ and checks equivalence relation only on boolean type. However, it performs subset check.
- $S4$: Number of executions of the inner while loop is $O(n^3)$, checks equivalence relation on all types and performs subset check.

7 Combining Pre-Condition, Subset Check, and Union Computations

All CTL operators can be defined in terms of a least or a greatest fixpoint. EF is defined as a least fixpoint as EF $p \equiv \mu x \,.\, p \;\vee\;$ EX x. Figure 3 shows the algorithm for computing the least fixpoint for EF. Given two composite formulas $P = \bigvee_{i=1}^{n_P} p_i$ and $R = \bigvee_{j=1}^{n_R} r_j$, where P represents a set of states and R represents the transition relation, the algorithm computes the set of states that satisfy $EF(P)$ iteratively. S_{new} represents the largest subset of the fixed point computed so far. At line 4 of *ComputeEF* algorithm $S_{new} = \bigvee_{k=1}^{n_S} s_k$ where one of the following holds for s_k:

1. $s_k = \text{PRE}(p_i, r_j)$, where $1 \leq i \leq n_P$ and $1 \leq j \leq n_R$, and $s_k \not\subseteq S_{old}$,
2. $s_k = \text{PRE}(p_i, r_j)$, where $1 \leq i \leq n_P$ and $1 \leq j \leq n_R$, and $s_k \subseteq S_{old}$,
3. $s_k \subseteq S_{old}$ and there exists no i, j, where $1 \leq i \leq n_P$ and $1 \leq j \leq n_R$, s.t $s_k = \text{PRE}(p_i, r_j)$.

Note that composite atoms that satisfy (1) can be used to decide if S_{new} is subset of S_{old} earlier during the computation of pre-condition and eliminate subset check at line 6 of the algorithm. This may serve as an improvement over the algorithm in Figure 3 since we can eliminate processing composite atoms in S_{new} that satisfy (3) during the subset check at line 6 of the algorithm. An additional improvement can be achieved by taking union of s_k with S_{res} only if s_k is not subset of S and prevent the unnecessary increase in the number of composite

PreUnion(S,R, *isSubset*): composite formula
S_{res}, S, R: composite formula
isSubset:boolean
$isSubset \leftarrow true$
$S_{res} \leftarrow S$
for each composite atom s in S **do**
  for each composite atom r in R **do**
    $s_k = \text{PRE}(s, r)$
    if $s_k \not\subseteq S$ **then**
      $isSubset \leftarrow false$
      $S_{res} \leftarrow S_{res} \cup s_k$
return S_{res}

(a)

EfficientEF(S, R): composite formula
S, R: composite formula
isSubset: boolean
$S_{new} \leftarrow S$
do
  $S_{old} \leftarrow S_{new}$
  $S_{new} \leftarrow PreUnion(S_{old}, R, isSubset)$
  $S_{new} \leftarrow Simplify(S_{new})$
while $(\neg isSubset)$
return S_{new}

(b)

Fig. 4. (a)A pre-condition algorithm with subset check and union (b) A more efficient algorithm for computing the least fixed point for EF

atoms in S_{res}. Figure 4(a) and 4(b) show the algorithms *preUnion*, which computes pre-condition along with subset check and union, and *EfficientEF*, which computes least fixed point for EF using *preUnion* algorithm, respectively.

8 Experiments

We have experimented the heuristics explained in the previous sections using a set of specifications. Some of the specifications are well known concurrency problems such as Bakery (BAKERY*) and Ticket (TICKET*) algorithms for mutual exclusion problem, Sleeping Barber (BARBER*), Readers Writers (RW*), and Bounded Buffer Producer Consumer (PC*) problems [And91]. We also included specifications of a cache coherence protocol (COHERENCE*) [DB], insertion sort (INSERTIONSORT) algorithm [DP01], an office light control system (LIGHTCONTROL) [BYK01] and a safety injection system for a nuclear reactor (SIS*) [CP93]. The "*"character indicates that the specification has several versions each with different temporal property or with different number of concurrent components. For Readers Writers and Sleeping Barber problems we have also used the parameterized versions meaning that the specification models the system with arbitrary number of processes. We obtained the experimental results on a SUN ULTRA 10 workstation with 768 Mbytes of memory, running SunOs 5.7.

Experimental results for the verifier with different versions of the simplification algorithm and without simplification is given in Table 1. The label $S2$-$S3$-$S4$ indicates that at each simplification point the simplification algorithm with $S2$, $S3$, and $S4$ are called in this order. So a multi-level simplification is achieved starting with the least aggressive version and continuing by increasing the degree of aggressiveness. Results show that multi-level simplification performs better than single level simplification. It also indicates that the speedup obtained by simplifying the composite representation is significant. When there is no simplification for most of the examples the verifier could not even complete due to memory blow up.

Problem Instance	S2-S3-S4 T	S2-S3-S4 M	S1 T	S1 M	S2 T	S2 M	S3 T	S3 M	S4 T	S4 M	None T	None M
bakery2-1	0.21	7.8	0.08	7.7	0.09	7.8	0.10	7.7	0.2	7.7	0.1	80
(L)bakery2-2	0.26	0.9	0.34	8.8	0.53	8.8	0.31	8.5	0.35	7.8	↑	↑
bakery3-1	8.26	19.6	3.61	21.2	3.44	21.5	3.48	20.3	8.85	19.5	5.71	30
(L)bakery3-2	51.32	34.7	255.45	324	370	323	109.7	77.5	81.23	34.9	↑	↑
ticket2-1	1.07	10.2	0.56	10.4	0.59	10.3	0.60	10.4	0.87	10.2	1.81	17.4
(L)ticket2-2	3.13	13.8	1.3	13.8	1.31	13.8	1.30	13.5	2.19	13.4	↑	↑
ticket3-1	14.71	28	15.39	58	15.49	58	13.56	43.3	21.42	35.2	↑	↑
(L)ticket3-2	29.73	29	61.28	173	75.48	173	28.2	62	119.95	60	↑	↑
barber2-1	4.62	17.7	4.8	21.5	4.52	21	3.59	17.6	4.52	17.6	59.3	197
barber2-2	0.27	8.8	0.21	9	0.25	9	0.23	9	0.27	8.8	0.28	9.3
barber2-3	1.58	12.8	1.2	13.7	1.49	13.8	1.51	13.8	1.55	12.8	2.93	20
barber3-1	10.93	26.6	13.68	35.2	13.22	34.5	9.14	26.4	10.59	26.4	↑	↑
barber3-2	0.35	9.5	0.35	9.8	0.32	9.7	0.33	9.8	0.30	9.5	0.5	10
barber3-3	4.12	18.1	2.79	20.5	4.39	21	4.31	21	3.93	18	66.89	205
barber4-1	21.98	38.5	26.86	53.7	29.92	52.7	18.85	38.1	21.05	38.1	↑	↑
barber4-2	0.43	10.1	0.43	10.5	0.46	10.5	0.43	10.5	0.39	10.1	0.83	13.3
barber4-3	8.76	25.9	5.31	30.7	10.09	32.2	9.93	32	8.61	25.8	↑	↑
barberp-1	6.69	24	5.17	21.1	5.64	24	6.53	24	7.8	24	173	228
barberp-2	0.21	9.3	0.13	9.3	0.16	9.3	0.19	9.3	0.20	9.3	0.38	10.3
barberp-3	1.19	13.4	0.72	12.4	0.78	13.4	0.89	13.4	1.14	13.4	3.77	19.5
coherence-1	0.34	11.3	0.24	11.2	0.25	11.2	0.29	11.2	0.30	11.3	0.23	11.5
(L)coherence-2	2.74	14.8	0.78	13.7	0.74	13.6	0.82	13.1	4.03	15.6	2.1	24.8
(L)coherence-3	11.97	29.8	2.09	22.8	2.11	22.2	2.42	21.1	18.8	32.8	21.97	161
(L)coherence-4	13.14	27.7	1.98	19.3	1.93	19.3	2.03	19.3	27.09	36.9	↑	↑
coherence-ref-1	0.27	11	0.27	11	0.19	11	0.22	11	0.25	11	0.16	11.1
(L)coherence-ref-2	0.97	11.4	>83	>60	>83	>60	>83	>60	>83	>60	↑	↑
(L)coherence-ref-3	5.52	20.8	>139	>181	>139	>181	>139	>181	>139	>181	↑	↑
(L)coherence-ref-4	23.25	27.7	>109	>118	>109	>118	>109	>118	>109	>118	↑	↑
insertionsort	0.2	8.5	0.12	8.5	0.11	8.5	0.11	8.4	0.23	8.4	0.21	8.7
pc5	0.07	7.7	0.05	7.7	0.06	7.7	0.06	7.7	0.05	7.7	0.05	8.1
pc10	0.09	8.5	0.09	8.5	0.08	8.5	0.10	8.5	0.09	8.5	0.09	9.5
pc30	0.25	11.8	0.24	11.8	0.25	11.8	0.25	11.8	0.23	11.8	0.23	28.3
rw16	0.02	8.1	0.02	8.1	0.02	8.1	0.02	8.1	0.02	8.1	↑	↑
rw32	0.03	10.8	0.03	10.8	0.03	10.8	0.03	10.8	0.03	10.8	↑	↑
rw64	0.05	20.6	0.05	20.6	0.05	20.6	0.05	20.6	0.05	20.6	↑	↑
rwp	0.01	9	0.01	9	0.01	9	0.01	9	0.01	9	0.01	9
sis-1	0.01	7.5	0.01	7.5	0.01	7.5	0.01	7.5	0.01	7.5	0.01	7.7
sis-2	0.07	19.4	0.02	19.4	0.02	19.4	0.03	19.4	0.06	19.4	0.02	23.8
lightcontrol	0.12	7.9	0.08	8	0.10	8	0.09	8	0.09	7.9	0.09	8.4

Table 1. Verification Time (T, in seconds) and Memory (M, in Mbytes) Results for Different Versions of Simplification vs No Simplification. (↑ means the program ran out of memory, $> x$ means execution did not terminate in x seconds, and (L) indicates that the specification has been verified for a liveness property)

The results in Table 2 show that combining subset check and union with pre-condition computation speedups the verification. There are two reasons for the speedup: 1) Since disjuncts that are computed as the result of the pre-condition computation are not included in the resulting composite formula if they are subset of the result from the previous iteration, the resulting composite formula has a smaller size. 2) Only the disjuncts, which are results of the pre-condition computation in the *current iteration*, are checked for subset relation against the resulting composite formula from the previous step. Average speedup for combining pre-condition computation with union and subset check heuristic is %24.

Verification times of the specifications with and without masking the integer pre-condition computation, if possible, by boolean satisfiability check are given in Table 2. Results show that masking integer pre-condition computation speeds up the verification and the speedup becomes higher for the specifications where the temporal property to be checked is a liveness property. The reason may be that liveness properties involve two fixpoint computations: one for EG and one for EF[1]. Additionally, a fixpoint iteration for EG involves a pre-condition computation followed by an intersection operation whereas a fixpoint iteration for EF involves a pre-condition computation followed by a union operation. Since intersection causes a quadratic increase (whereas union causes a linear increase) in the composite formula size, EG fixpoint iterates are likely to grow faster. Average speedup for masking heuristic is %13.

Verification times of the specifications for inefficient and efficient subset check algorithms are given in Table 2. In most of the experiments the efficient subset check algorithm shown in Figure 1(b) performs better than the subset check algorithm shown in Figure 1(a) that computes negation operation on composite formula level. As a result it supports the idea behind the efficient subset check algorithm: Process composite atoms as needed and perform negation on composite atom level instead of composite formula level which has an exponential time complexity. Average speedup for efficient subset check heuristic is %11.

9 Conclusion

We presented several heuristics for efficient manipulation of composite symbolic representation that combines several constraint representations that are specialized to different domains. We have implemented these heuristics and used them in Composite Symbolic Library which is a symbolic manipulator for model checking systems with heterogeneous data.

Our experiments indicate that verification times are reduced significantly by using our simplification heuristic for composite formulas. When simplification was not used approximately %50 of the examples could not even complete since they ran out memory. Heuristics for combining the pre-condition computation with subset check and union, avoiding the negation at composite formula level and performing it at composite atom level, and masking expensive integer manipulation with boolean manipulation also improve the verification times significantly. Composite Symbolic Library is available at:
http://www.cs.ucsb.edu/˜bultan/composite/

References

[AHH96] R. Alur, T. A. Henzinger, and P. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[1] Note that the verifier computes the negation of a liveness property $AGAFp$ and computes the fixpoint for $EFEG(\neg p)$. Then it checks satisfiability of $I \wedge EFEG(\neg p)$. Similarly, for an invariant property AGp it computes the fixpoint for the negation of AGp and checks satisfiability of $I \wedge EF(\neg p)$

[And91] G. R. Andrews. *Concurrent Programming: Principles and Practice.* The Benjamin/Cummings Publishing Company, Redwood City, California, 1991.

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

[BGL00] T. Bultan, R. Gerber, and C. League. Composite model checking: Verification with type-specific symbolic representations. *ACM Transactions on Software Engineering and Methodology*, 9(1):3–50, January 2000.

[BGP99] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, July 1999.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BYK01] T. Bultan and T. Yavuz-Kahveci. Action language verifier. In *Proceedings of the 6th IEEE International Conference on Automated Software Engineering (ASE 2001)*, 2001.

[CAB+98] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[CGP99] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model checking.* The MIT Press, Massachusetts, Cambridge, 1999.

[CP93] P. J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proceedings of the 15th International Conference on Software Engineering*, pages 315–323, May 1993.

[CUD] CUDD: CU decision diagram package, http://vlsi.colorado.edu/ fabio/cudd/.

[DB] G. Delzanno and T. Bultan. Constraint-based verification of client server protocols. In *Proceedings of the Seventh International Conference on Principles and Practice of Constraint Programming (CP 2001).*

[DP01] Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *Journal of Software and Tools for Technology Transfer*, 3(3):250–270, 2001.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In C. Courcoubetis, editor, *Proceedings of computer aided verification*, volume 697 of *Lecture Notes in Computer Science*, pages 333–346. Springer-Verlag, 1993.

[HRP94] N. Halbwachs, P. Raymond, and Y. Proy. Verification of linear hybrid systems by means of convex approximations. In B. LeCharlier, editor, *Proceedings of International Symposium on Static Analysis*, volume 864 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1994.

[McM93] K. L. McMillan. *Symbolic model checking.* Kluwer Academic Publishers, Massachusetts, 1993.

[Ome] The Omega project, http://www.cs.umd.edu/projects/omega/.

[Sri93] D. Srivastava. Subsumption and indexing in constraint query languages with linear arithmetic constraints. *Annals of Mathematics and Artificial Intelligence*, 8:315–343, 1993.

[YKTB01] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. Composite symbolic library. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, April 2001.

Problem Instance	−Subset T	−Subset M	+Subset T	+Subset M	−PreUnion T	−PreUnion M	+PreUnion T	+PreUnion M	−Mask T	−Mask M	+Mask T	+Mask M
BAKERY2-1	0.26	10.9	0.21	7.8	0.32	7.8	0.21	7.8	0.24	8.7	0.21	7.8
(L)BAKERY2-2	0.34	9.4	0.26	7.9	0.3	7.9	0.26	7.9	0.41	8	0.26	7.9
BAKERY3-1	20.99	268	8.26	19.6	20.37	16.1	8.26	19.6	8.34	28.8	8.26	19.6
(L)BAKERY3-2	54	69	51.32	34.7	50.04	34.7	49.47	34.7	57.34	36.6	51.32	34.7
TICKET2-1	1.16	18.3	1.07	10.2	1.22	9.6	1.07	10.2	1.14	13.2	1.07	10.2
(L)TICKET2-2	3.31	23.2	3.13	13.8	3.26	12.3	3.13	13.8	3.35	19.7	3.13	13.8
TICKET3-1	18.87	159	14.71	28	29.14	20.1	14.71	28	14.76	37.4	14.71	28
(L)TICKET3-2	36.47	204	29.73	29	66.53	34.2	29.73	29	32.67	49.9	29.73	29
BARBER2-1	8.03	122	4.62	17.7	6.32	15.7	4.62	17.7	4.65	21.4	4.62	17.7
BARBER2-2	0.2	10.1	0.27	8.8	0.37	8.3	0.27	8.8	0.24	9	0.2	8.8
BARBER2-3	1.69	34.4	1.58	12.8	2	11.1	1.58	12.8	1.63	14.5	1.58	12.8
BARBER3-1	21.75	299	10.93	26.6	11.53	18.8	10.93	26.6	11.07	31.1	10.93	26.6
BARBER3-2	0.21	11.2	0.35	9.5	0.46	8.6	0.35	9.5	0.36	9.7	0.35	9.5
BARBER3-3	6.03	105	4.12	18.1	4.15	13.7	4.12	18.1	4.12	20.5	4.12	18.1
BARBER4-1	43.77	554	21.98	38.5	20.62	22.8	21.98	38.5	22.21	44.4	21.98	38.5
BARBER4-2	0.27	12.3	0.43	10.1	0.59	8.9	0.43	10.1	0.44	10.4	0.43	10.1
BARBER4-3	16.94	264	8.76	25.9	8.38	16.7	8.76	25.9	8.87	29.1	8.76	25.9
BARBERP-1	3	23.1	2.83	12.9	4.49	12.9	2.83	12.9	2.9	14.9	2.83	12.9
BARBERP-2	0.3	10.4	0.29	9.2	0.6	9.7	0.29	9.2	0.32	9.5	0.29	9.2
BARBERP-3	3.04	37.4	2.62	14.5	4.13	13.8	2.62	14.5	2.72	17.6	2.62	14.5
COHERENCE-1	0.37	13.4	0.34	11.3	0.86	10	0.34	11.3	0.38	13	0.34	11.3
(L)COHERENCE-2	4.92	53.9	2.74	14.8	4.88	15.4	2.74	14.8	3.43	29.3	2.74	14.8
(L)COHERENCE-3	21.61	169	11.97	29.8	26.45	35.4	11.97	29.8	13.93	91.1	11.97	29.8
(L)COHERENCE-4	15.65	96.2	13.14	24.6	13.38	22.1	13.14	24.6	25.64	49.7	13.14	24.6
COHERENCE-REF-1	0.28	11.9	0.27	11	0.56	10.1	0.27	11	0.33	12.7	0.27	11
(L)COHERENCE-REF-2	1.17	21.2	0.97	11.4	1.93	11.2	0.97	11.4	1.26	14.3	0.97	11.4
(L)COHERENCE-REF-3	6.47	67.5	5.5	20.8	5.62	15.4	5.52	20.8	6.79	42.4	5.52	20.8
(L)COHERENCE-REF-4	27.45	125	23.25	27.7	26.52	29.5	23.25	27.7	49.38	49.4	23.25	27.7
INSERTIONSORT	0.2	9.1	0.2	8.5	0.26	8.6	0.2	8.5	0.3	10.8	0.2	8.5
PC5	0.07	7.6	0.07	7.7	0.12	7.7	0.07	7.7	0.07	7.8	0.07	7.7
PC10	0.09	8.4	0.09	8.5	0.22	8.5	0.09	8.5	0.09	8.6	0.09	8.5
PC30	0.25	11.6	0.25	11.8	0.62	11.7	0.25	11.8	0.26	11.8	0.25	11.8
RW16	0.02	8.1	0.02	8.1	0.03	8.1	0.02	8.1	0.02	8.1	0.02	8.1
RW32	0.03	10.8	0.03	10.8	0.05	10.8	0.03	10.8	0.04	10.8	0.03	10.8
RW64	0.04	20.6	0.05	20.6	0.1	20.6	0.05	20.6	0.08	20.6	0.05	20.6
RWP	0.01	9	0.01	9	0.01	9	0.01	9	0.01	9	0.01	9
SIS-1	0.01	7.5	0.01	7.5	0.01	7.5	0.01	7.5	0.01	7.5	0.01	7.5
SIS-2	0.07	19.4	0.07	19.4	0.08	19.4	0.07	19.4	0.38	27.2	0.07	19.4
LIGHTCONTROL	0.13	8.8	0.12	7.9	0.17	7.6	0.12	7.9	0.12	8.5	0.12	7.9

Table 2. Verification Time (T, in seconds) and Memory (M, in Mbytes) Results for Inefficient vs Efficient Subset Check, Pre-Condition Computation Without Union vs With Union, and Without Integer Masking vs With Integer Masking.(L) indicates that the specification has been verified for a liveness property. + and - denote inclusion and exclusion of the heuristic, respectively.