Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

A Symbolic Manipulator for Automated Verification of Reactive Systems with Heterogeneous Data Types

Tuba Yavuz-Kahveci and Tevfik Bultan

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

Received: date / Revised version: date

Abstract. In this paper, we present design and implementation of the *Composite Symbolic Library*, a symbolic manipulator for model checking systems with heterogeneous data types. Our tool provides a common interface for different symbolic representations, such as BDDs for representing boolean logic formulas and polyhedral representations for linear arithmetic formulas. Based on this common interface, these data structures are combined using a disjunctive composite representation. We propose several heuristics for efficient manipulation of this composite representation and present experimental results that demonstrate their performance. We used an object-oriented design to implement the Composite Symbolic Library. We imported the CUDD library (a BDD library) and the Omega Library (a linear arithmetic constraint manipulator that uses polyhedral representations) to our tool by writing wrappers around them which conform to our symbolic representation interface. Our tool supports polymorphic verification procedures which dynamically select symbolic representations based on the input specification. Our symbolic representation library can be used as an interface between different symbolic libraries, model checkers, and specification languages. We expect our tool to be useful in integrating different tools and techniques for symbolic model checking, and in comparing their performance.

1 Introduction

Compact and efficient symbolic representations have 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 logic formulas) and polyhedral representation [Hal93] (for representing linear arithmetic formulas) 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 the 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,DP01,HRP94] which cannot be verified 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.

Generally, model checking tools have been built using a single symbolic representation [McM93,AHH96]. The representation used depends on the target application domain for the model checker. Inefficiencies of the symbolic representation used in a model checker can be addressed using various abstraction techniques, some ad hoc, such as restricting variables to finite domains, some formal, such as predicate-abstraction [BLO98,Sai00,BPRue]. These abstraction techniques can be used independent of the symbolic representation. As model checkers become more widely used, it is not hard to imagine that a user would like to use a model checker built for real-time systems on a system with lots

This work is supported in part by NSF grant CCR-9970976 and NSF CAREER award CCR-9984822.

of boolean variables and only a couple of real variables. Similarly another user may want to use a BDD-based model checker to check a system with few boolean variables but lots of integer variables. Currently, such users may need to obtain a new model-checker for these instances, or use various abstraction techniques to solve a problem which may not be suitable for the symbolic representation their model checker is using. More importantly, as symbolic model-checkers are applied to larger problems, they are bound to encounter specifications with different variable types which may not be efficiently representable using a single symbolic representation.

In this paper we present a verification tool which combines different symbolic representations instead of using a single symbolic representation. Different symbolic representations are combined using the *composite model checking* approach presented in [BGL98, BGL00b]. Each variable type in the input specification is assigned to the most efficient representation for that variable type. The goal is to have a platform where strength of each symbolic representation is utilized as much as possible, and deficiencies of a representation are compensated by the existence of other representations.

The main contributions of this work can be outlined as follows:

- *Tool Design:* Our tool is designed using the object-oriented paradigm. The heart of the Composite Symbolic Library is a common interface that abstracts the functionality of a symbolic representation [YKTB01]. All symbolic representations are defined as classes derived from this interface. Different symbolic libraries can be integrated to our tool by writing wrappers around them which implement this interface. The object-oriented design provides the following advantages: 1) the Composite Symbolic Library can be easily extended with new symbolic representations, 2) verification procedures interact with different symbolic representation libraries using a single interface, and 3) verification procedures are polymorphic, i.e., the verifier decides which symbolic representations to use at run-time.
- *Algorithms and Complexity Analysis:* We present algorithms and their complexity analysis for the manipulation of the composite symbolic representation. We define the time complexities of the algorithms in terms of the time complexities of the operations on the basic symbolic representations.
- *Heuristics:* We present heuristics for efficient manipulation of the composite symbolic 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 interleaving the computation of pre and post-conditions with subset checks, and 3) the size of a composite representation can be minimized by iteratively merging matching constraints and removing redundant ones. We experimented on a large set of examples to show the effectiveness of our heuristics.

Related Work. There have been other studies which combine different symbolic representations. In [CABN97], Chan *et al.* present a technique in which (both linear and non-linear) constraints are mapped to BDD variables and a constraint solver is used during model checking computations (in conjunction with SMV) to prune infeasible combinations of these constraints. Although this technique is capable of handling non-linear constraints, it is restricted to systems where transitions are either *data-memoryless* (i.e., next state value of a data variable does not depend on its current state value), or *data-invariant* (i.e., data variables remain unchanged). Hence, even a transition which increments a variable (i.e., $x' = x + 1$) is ruled out. It is reported in [CABN97] that this restriction is partly motivated by the semantics of RSML, and it allows modeling of a significant portion of TCAS II system.

In [BS00], a tool for checking inductive invariants on SCR specifications is described. This tool combines automata based representations for linear arithmetic constraints with BDDs. This approach is similar to our approach but it is specialized for inductive invariant checking. Another difference is, our tool uses polyhedral representations as opposed to automata based representations for linear arithmetic. However, because of the object-oriented design of our tool it should be easy to extend it with automata-based linear constraint representations.

The Symbolic Analysis Laboratory provides a framework for combining different tools in verification of concurrent systems [BGL$^+$00a]. The heart of the Symbolic Analysis Laboratory is a language for specifying concurrent systems in a compositional manner. Our Composite Symbolic Library is a low-level approach compared to the Symbolic Analysis Laboratory. We are combining different libraries at the symbolic representation level as opposed to developing a specification language to integrate different tools.

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 the composite formulas which contain a mixture of boolean and integer variables. In Hytech boolean and 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 a composite representation.

The rest of the paper is organized as follows. In Section 2 we define the composite symbolic representation and describe the architecture and the design of the Composite Symbolic Library. We present the algorithms for manipulating the composite symbolic representation in Section 3. In Section 4 we describe some heuristics for improving the performance of the Composite Symbolic Library. In Sections 5 and 7 we compare the performance of our model checker with the Omega Library Model Checker [BGP97,BGP99] which uses only polyhedral representation. We present the experiments that demonstrate the effectiveness of our heuristics in Section 6. Finally, in Section 8 we conclude and give some future directions for our work.

2 Composite Symbolic Representation

To combine different symbolic representations we use the composite model checking approach presented in [BGL98,BGL00b]. The basic idea in composite model checking is to map each variable in the input specification to a symbolic representation type. For example, boolean and enumerated variables can be mapped to the BDD representation, and integers can be mapped to an arithmetic constraint representation. We encode the sets of system states and transitions as a disjunction of conjunctions of type specific representations. For example, a disjunct may consist of a boolean formula stored as a BDD representing the states of boolean and enumerated variables, and a linear arithmetic constraint representation representing the states of integer variables. We call this disjunctive representation a *composite representation.* Each atomic event in the input specification is conjunctively partitioned where each conjunct specifies the effect of the event on the variables represented by a single symbolic representation. For example, one conjunct specifies the effect of the event on variables encoded using BDDs, whereas another conjunct specifies the effects of the event on variables encoded using linear arithmetic constraints. The pre- and post-condition computations are computed independently for each symbolic representation by exploiting the conjunctive partitioning of the atomic events. The key observation here is the fact that conjunctive partitioning of the atomic events allows pre and post-condition computations to distribute over different symbolic representations. We also implement algorithms for intersection, union, complement, and subset, equivalence and emptiness checking computations for the disjunctive composite representation which use the corresponding methods for different symbolic representations.

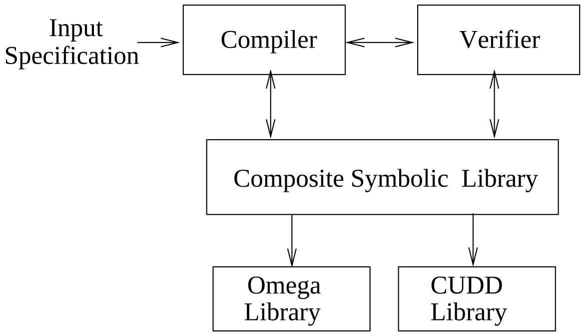

Fig. 1. Architecture of the composite model checker

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 *basic symbolic representations*. For the BDD representations we use the Colorado University Decision Diagram Package (CUDD) [CUD]. For the Presburger arithmetic formula manipulation we use the Omega Library [KMP+95, Ome]. Fig. 1 illustrates the general architecture of our composite model checking system. We will focus on the Composite Symbolic Library in this paper.

We implemented the Composite Symbolic Library in C++ and Fig. 2 shows its class hierarchy as a UML class diagram[1]. The abstract class `Symbolic` serves as an interface to all symbolic representations including the composite representation. Our current specification language supports enumerated, boolean, and integer variables. Our system maps enumerated variables to boolean variables. The classes `BoolSym` and `IntSym` are the symbolic representations for boolean and integer variable types, respectively. The class `BoolSym` serves as a wrapper for the BDD library CUDD [CUD]. It is derived from the abstract class `Symbolic`. Similarly, `IntSym` is also derived from the abstract class `Symbolic` and serves as a wrapper for the Omega Library [Ome].

The class `CompSym` is the class for composite representation. It is derived from `Symbolic` and uses `IntSym`

[1] In UML class diagrams, triangle arcs denote *generalization*, diamond arcs denote *aggregation*, dashed arcs denote *dependency*, and solid lines denote *association* among classes.

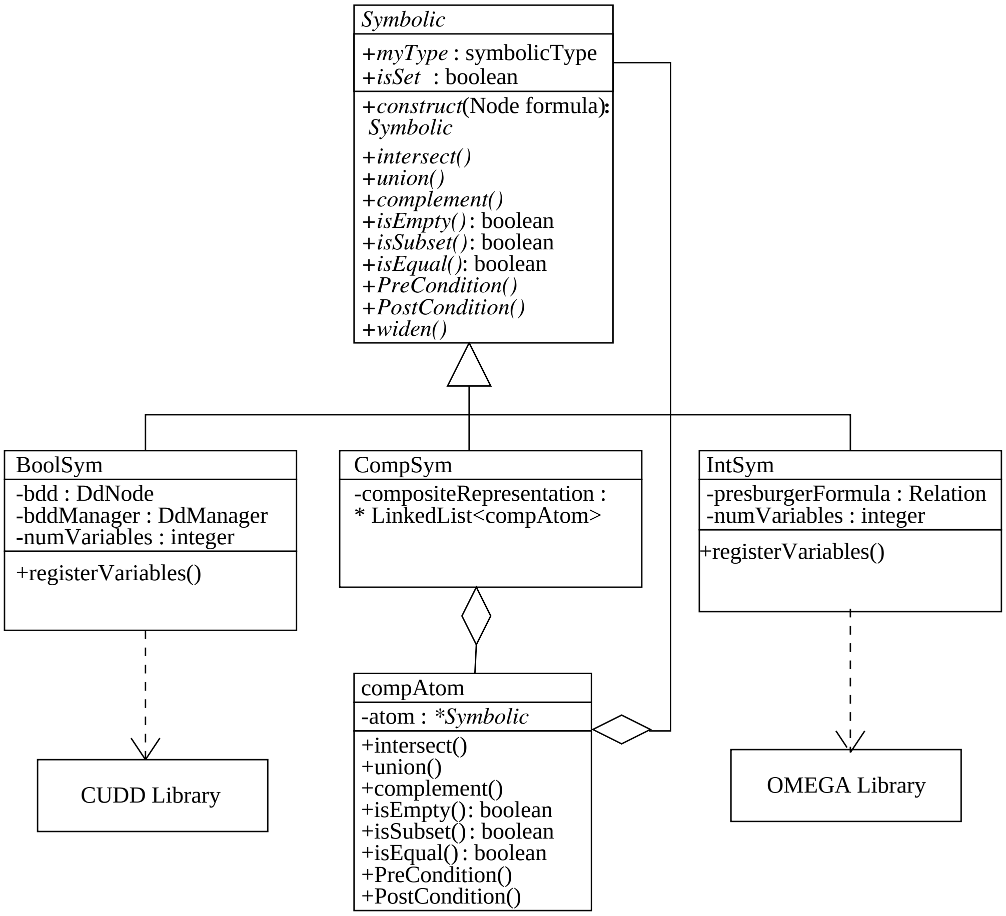

Fig. 2. Class diagram for the Composite Symbolic Library

and `BoolSym` (through the `Symbolic` interface) to manipulate the composite representation. There is no dependency among the `CompSym` class and the `IntSym` and the `BoolSym` classes. Note that this design is an instance of the composite design pattern given in [GHJV94].

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 the number of basic symbolic representations. The `CompSym` class accesses the 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 the 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 verify a system with our tool, 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 integer arithmetic formulas on integer variables with boolean variables using logical connectives. The syntax of a composite formula is defined as follows:

$$\begin{aligned}
CF &::= CF \wedge CF \mid CF \vee CF \mid \neg CF \mid BF \mid IF \\
BF &::= BF \wedge BF \mid BF \vee BF \mid \neg BF \mid Term_{bool} \\
IF &::= IF \wedge IF \mid IF \vee IF \mid \neg IF \\
&\quad \mid Term_{int}\ Rop\ Term_{int} \\
Term_{bool} &::= id_{bool} \mid true \mid false \\
Term_{int} &::= Term_{int} + Term_{int} \mid Term_{int} - Term_{int} \\
&\quad \mid - Term_{int} \mid id_{int} \mid constant \\
&\quad \mid id_{int} \times constant
\end{aligned}$$

where CF, BF, IF, and Rop denote composite formula, boolean formula, integer formula, and relational operator ($>$, $<$, $\geq$, $\leq$, $=$, $\neq$), respectively. Since the symbolic representations in our Composite Symbolic Library currently support only boolean and linear arithmetic formulas, we restrict arithmetic operators to $+$ and $-$, but we allow multiplication with a constant. In the future, by adding new symbolic representations we can extend this grammar.

A transition relation can be specified using a composite formula by using unprimed variables to denote the current state variables and primed variables to denote the next state variables. A method called `registerVariables` in `BoolSym` and `IntSym` is used to register the current and next state variable names during the initialization of the representation.

Given a composite formula, the method `construct()` in the `Symbolic` class traverses the syntax tree and calls constructor of the `BoolSym` class when a boolean formula is encountered and calls constructor of the `IntSym` class when an integer formula is encountered. If the composite formula consists of both integer and boolean formulas then constructor of the `CompSym` class is called. In the `CompSym` class, a composite formula, A, is represented in our *composite representation* as

$$A \equiv \bigvee_{i=1}^{n} \bigwedge_{t \in T} a_{it}$$

where a_{it} denotes the formula of type t in the ith disjunct, and n and T denote the number of disjuncts and the set of basic symbolic representations, respectively.

We call each disjunct $\wedge_{t \in T} a_{it}$ a *composite atom*. Fig. 3 shows the composite atoms in an example composite formula. Each composite atom is implemented as an instance of a class called `compAtom` (see Fig. 2). Each `compAtom` object represents a conjunction of formulas each of which is either a boolean or an integer formula.

A composite formula stored in a `CompSym` object is implemented as a list of `compAtom` objects, which corresponds to the disjunction in the composite representation. Note that `Symbolic` members of the `compAtom` class cannot be of type `CompSym`. Fig. 4 shows internal representation of the composite formula given in Fig. 3 in a `CompSym` object. The field *atom* is an array of pointers to the class `Symbolic` and the size of the array is the number of basic symbolic representations.

The `CompSym` and the `compAtom` classes use a `TypeDescriptor` class which records the variable types used in the input specification. Our library can adapt itself to any subset of the supported variable types, i.e., if a variable type is not present in the input specification, the symbolic library for that type will not be called during the execution. For example, given an input specification with no integer variables our tool will behave as a BDD-based model checker without making any calls to the Omega Library.

3 Manipulation of the Composite Representation

In this section we present algorithms for basic set operations and pre- and post-condition computations on our disjunctive composite representation. These algorithms are implemented as methods in the `compAtom` and the `CompSym` classes in the Composite Symbolic Library. Note that the algorithms given in this section are independent of the type and the number of basic symbolic representations used.

IsSubset(composite atom a, composite atom b) : boolean
  for each basic symbolic representation t do
    if $a_t \not\subseteq b_t$ then
      return false
  return true

Fig. 5. Algorithm for checking the subset relation between two composite atoms

Throughout this section, n_A, T, and T^t_{Op} denote the number of composite atoms in composite formula A, the set of basic symbolic representations in the Composite Symbolic Library, and the time complexity of the operation Op for the basic symbolic representation t. The operations are $Op \in \{Intersection, Union, Complement, IsSubset, IsEqual, IsEmpty\}$. Note that we interpret each composite representation as the set of valuations that satisfy the corresponding composite formula.

Subset Check: A composite atom $a \equiv \bigwedge_{t \in T} a_t$ is subset of a composite atom $b \equiv \bigwedge_{t \in T} b_t$ iff for each symbolic representation a_t in a, a_t is subset of b_t, which is the corresponding symbolic representation in b. For instance let composite atoms a and b be

$$a \equiv x \wedge y \wedge z > 0, b \equiv x \wedge z \geq 0$$

where x and y are boolean variables and z is an integer variable. $a \subseteq b$ since the valuations of the variables x and y which satisfy the formula $x \wedge y$ is subset of the valuations of the variables x and y which satisfy the formula x, and the valuations of the variable z which satisfy the formula $z > 0$ is subset of the valuations of the variable z which satisfy the formula $z \geq 0$. Fig. 5 shows the *IsSubset* algorithm for checking subset relation between two composite atoms. The worst case time complexity of the algorithm is $O(\sum_{t \in T} T^t_{IsSubset})$.

A composite formula $A \equiv \bigvee_{i=1}^{n_A} a_i$ is subset of a composite formula $B \equiv \bigvee_{k=1}^{n_B} b_k$ iff $\forall i$ s.t. $1 \leq i \leq n_A$, $a_i \subseteq B$. For instance let A and B be

$$A \equiv (x \wedge z > 0) \vee (x \wedge y \wedge z \leq 0), B \equiv z \geq 0 \vee x$$

where x and y are boolean variables and z is an integer variable. A is a subset of B since both composite atoms $(x \wedge z > 0)$ and $(x \wedge y \wedge z \leq 0)$ are subsets of B. Note that $(x \wedge z > 0)$ is subset of $z \geq 0$ and $(x \wedge y \wedge z \leq 0)$ is subset of x. So the most straightforward way of checking the subset relation between two composite formulas A and B is to iterate through the composite atoms in A and check the 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 a subset of B we can conclude that A

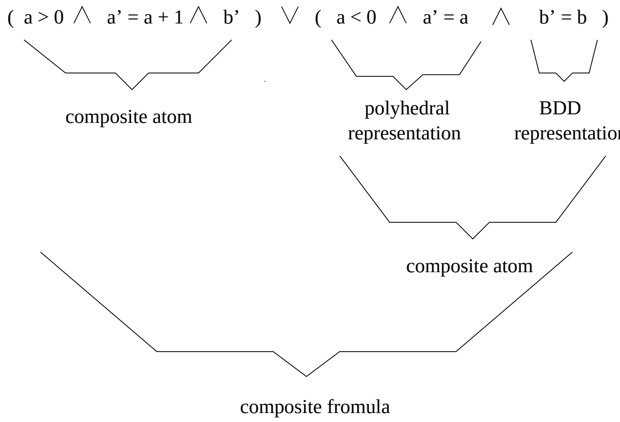

Fig. 3. An example composite formula. a is an integer variable and b is a boolean variable.

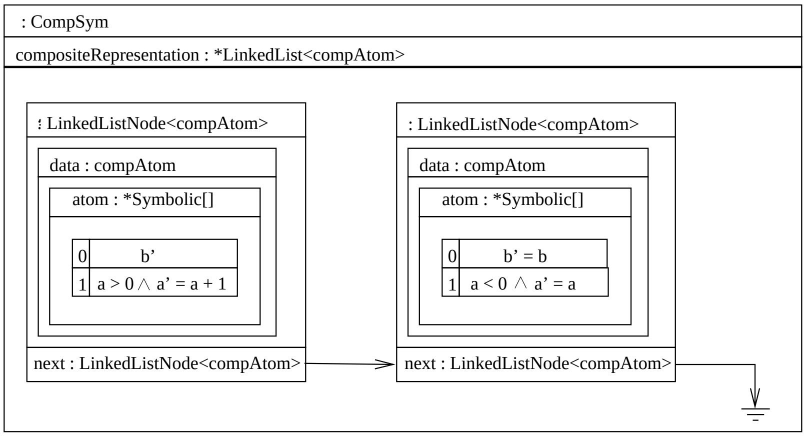

Fig. 4. An instance of the `CompSym` class representing the composite formula in Fig. 3

is not a subset of B. On the other hand, if there exists no such composite atom in A then we can conclude that A is a subset of B.

Fig. 6 shows the algorithm for checking the 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 a subset of any composite atom in B (lines 5-9). If there exists no composite atom b in B such that a is a subset of b then this does not mean that a is not a subset of B. Next, the algorithm computes $a \wedge \neg B$ and assigns the result to a composite formula C. If C is not empty, then this means that a is not a subset of B and the algorithm exits by returning false (lines 10-13). Otherwise, the algorithm continues until either it finds out that there exists a composite atom a that is not a subset of B or it has checked all the composite atoms in A, in which case it returns true (line 14). The worst case time complexity of the algorithm is

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

```
1.  IsSubset(A, B): boolean
2.      found: boolean
3.      A, B, C: composite formula
4.      for each composite atom a in A do
5.          found ← false
6.          for each composite atom b in B do
7.              if a ⊆ b then
8.                  found ← true
9.                  break
10.         if ¬found then
11.             C ← a ∧ ¬B
12.             if C ≢ ∅ then
13.                 return false
14.     return true
```

Fig. 6. Algorithm for checking the subset relation between two composite formulas

The exponential component of the formula is due to the complement operation at line 11 in Fig. 6, which is of exponential time complexity as it will be explained below.

```
IsEmpty(composite atom a): boolean
    for each symbolic representation t do
        if a_t ≡ ∅ then
            return true
    return false
```

Fig. 7. Algorithm for checking emptiness of a composite atom

```
IsEmpty(composite formula A): boolean
    for each composite atom a in A do
        if a ≢ ∅ then
            return false
    return true
```

Fig. 8. Algorithm for checking emptiness of a composite formula

Equivalence Check: A composite atom a is equivalent to a composite atom b iff each symbolic representation a_t in a is equivalent to b_t which is the corresponding symbolic representation in b. A composite formula A is equivalent to a composite formula B iff A is a subset of B and B is a subset of A. The worst case time complexity of checking the equivalence of two composite atoms is $O(\sum_{t \in T} T^t_{IsEquivalent})$ and that of checking the equivalence of two composite formulas is $O(max(T_{IsSubset(A,B)}, T_{IsSubset(B,A)}))$.

Emptiness Check: A composite atom a is empty iff there exists a symbolic representation a_t in a such that a_t is empty. Fig. 7 shows *IsEmpty* algorithm for checking emptiness of a composite atom. The worst case time complexity of the algorithm is $O(\sum_{t \in T} T^t_{IsEmpty})$.

A composite formula A is empty iff for all composite atoms a_i in A, a_i is empty. Fig. 8 shows *IsEmpty* algorithm for a composite formula. The worst case time complexity of the algorithm is $O(n_A \times \sum_{t \in T} T^t_{IsEmpty})$.

Pre-Condition Computation: Given two composite formulas $A \equiv \bigvee_{i=1}^{n_A} \bigwedge_{t \in T} a_{it}$ and $B \equiv \bigvee_{k=1}^{n_B} \bigwedge_{t \in T} b_{kt}$, where A represents the set of states and B represents the transition relation, the pre-condition of A with respect to B can be computed as

$$Pre(A,B) \equiv \bigvee_{i=1}^{n_A} \bigvee_{k=1}^{n_B} \bigwedge_{t \in T} \exists V' a'_{it} \wedge b_{kt}$$

where V' is the set of next state variables and a_{it}' is obtained by replacing every variable in a_{it} with the corresponding next-state variable. Note that the above property holds because the existential variable elimination in the $Pre(A,B)$ 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 [BGL00b].

Computing the pre-condition of A with respect to B is equivalent to computing the set of states that can reach the set of states represented by A by a single transition in B. For instance let A and B be

$$A \equiv y = 1, \;\; B \equiv (x \wedge y' = 1) \vee (\neg x \wedge y \geq 0 \wedge y' = y + 1)$$

where x is a boolean variable and y is an integer variable. Then $Pre(A,B)$ can be computed as:

$$\begin{aligned} Pre(A,B) &\equiv Pre(y = 1, x \wedge y' = 1) \vee \\ &\quad\; Pre(y = 1, \neg x \wedge y \geq 0 \wedge y' = y + 1) \\ &\equiv x \vee (\neg x \wedge y = 0) \end{aligned}$$

```
Pre(composite atom a, composite atom b): composite atom
    c: composite atom
    for each symbolic representation t do
        c_t ← ∃V' a'_t ∧ b_t
    return c
```

Fig. 9. Algorithm for computing the pre-condition of a composite atom with respect to a composite atom

```
Pre(composite formula A, composite formula B): composite formula
    C: composite formula
    for each composite atom a in A do
        for each composite atom b in B do
            C ← C∨ Pre(a,b)
    return C
```

Fig. 10. Algorithm for computing the pre-condition of a composite formula with respect to a composite formula

Fig. 9 and 10 show the pre-condition computation algorithms for composite atoms and composite formulas, respectively. The worst case time complexity of the pre-condition algorithm for composite atoms is $O(\sum_{t \in T} T^t_{Pre\text{-}Condition})$ and that of the pre-condition algorithm for composite formulas is $O(n_A \times n_B \times \sum_{t \in T} T^t_{Pre\text{-}Condition})$.

Intersection: Given two composite formulas, A and B, their intersection can be computed as:

$$A \wedge B \equiv \bigvee_{i=1}^{n_A} \bigvee_{k=1}^{n_B} \bigwedge_{t \in T} (a_{it} \wedge b_{kt})$$

The worst case time complexity of computing the intersection of two composite atoms is $O(\sum_{t \in T} T^t_{Intersection})$ and that of computing the intersection of two composite formulas is $O(n_A \times n_B \times \sum_{t \in T} T^t_{Intersection})$.

Complement : The complement of a composite atom $a \equiv \bigwedge_{t \in T} a_t$ is a composite formula $B \equiv \bigvee_{t \in T} \neg a_t$. Given a composite formula A we can compute A's complement as

$$\neg A \equiv \bigvee_{i=1}^{|T|^{n_A}} a_i$$

where

$$a_i \equiv \bigwedge_{j=1, t_j \in T}^{n_A} \neg a_{t_j}.$$

Complement(composite atom a): composite formula
  A: composite formula
  $A \leftarrow false$
  for each symbolic representation a_t in a do
    $A \leftarrow A \vee \neg a_t$
  return A

Fig. 11. Algorithm for computing the complement of a composite atom

Complement(composite formula A): composite formula
  B: composite formula
  $B \leftarrow false$
  let $A \equiv \bigvee_{i=1}^{n_A} \bigwedge_{t \in T} a_{it}$
  for each combination of $(a_{1_{t_1}}, a_{2_{t_2}}, ..., a_{n_{t_n}})$ s.t. $t_i \in T$ and $1 \leq i \leq n_A$ do
    $B \leftarrow B \vee \bigwedge_{i=1}^{n_A} \neg a_{i_{t_i}}$
  return B

Fig. 12. Algorithm for computing the complement of a composite formula

Fig. 11 and 12 show the complement algorithms for a composite atom and a composite formula, respectively. The worst case time complexity of the complement algorithm for a composite atom is $O(\sum_{t \in T} T^t_{Complement})$ and that of the complement algorithm for a composite formula is

$$O(\sum_{i=1}^{n_A} \sum_{t \in T} T^t_{Complement} + |T|^{n_A} \times \sum_{j=1, t_j \in T}^{n_A} T^{t_j}_{Intersection}).$$

Union: The union of a composite atom a with a composite atom b is a composite formula $A \equiv a \vee b$. Given two composite formulas, A and B, we define A union B as

$$A \vee B \equiv \bigvee_{i=1}^{n_A+n_B} \bigwedge_{t \in T} c_{it}$$

where for $1 \leq i \leq n_A$ $c_{it} \equiv a_{it}$ and for $n_A + 1 \leq i \leq n_A + n_B$ $c_{it} \equiv b_{it}$.

The worst case time complexity of computing the union of two composite atoms is constant and that of computing the union of two composite formulas is $O(n_A + n_B)$. As the worst case time complexity formulas show, while computing the union of two composite atoms or two composite formulas, no operation is performed on symbolic representation level. Since we use a disjunctive composite representation, the union operation can be performed by a concatenation operation on the linked list structure which is used to implement a `CompSym` object (see Fig. 4).

4 Heuristics for Efficient Automated Verification with the Composite Representation

In this section we present several heuristics which improve the performance of automated verification using the Composite Symbolic Library [YKB02]. Our heuristics make use of the following observations: 1) efficient operations on BDDs (e.g., satisfiability checking) can be used to mask expensive operations on polyhedra (e.g., image computations and satisfiability checking), 2) our disjunctive representation can be exploited by interleaving the computation of pre and post-conditions with subset checks, and 3) the size of a composite representation can be minimized by iteratively merging matching constraints and removing redundant ones. In the following subsections we present these heuristics in detail.

4.1 Masking Integer Operations

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

A composite atom $a \equiv a_b \wedge a_i$, where b and i denote the BDD part and the polyhedral part, respectively, is satisfiable iff both a_b and a_i are satisfiable. Since satisfiability check for the polyhedral representation is expensive, by checking the satisfiability of the BDD part first we can avoid checking the 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 $A \equiv \bigvee_{j=1}^{n_A} a_{jb} \wedge a_{ji}$ and $C \equiv \bigvee_{k=1}^{n_C} c_{kb} \wedge c_{ki}$, where A represents a set and C represents the transition relation, a_{jb} and c_{kb} correspond to boolean formulas, and a_{ji} and c_{ki} correspond to integer formulas, pre-condition of A with respect to C can be written as

$$Pre(A, C) \equiv \bigvee_{j=1}^{n_A} \bigvee_{k=1}^{n_C} Pre(a_{jb}, c_{kb}) \wedge Pre(a_{ji}, c_{ki})$$

Instead of computing $Pre(a_{jb}, c_{kb})$ and $Pre(a_{ji}, c_{ki})$ and then taking the intersection of the two, we can first compute $Pre(a_{jb}, c_{kb})$ and then check it for satisfiability. Since $Pre(a_{jb}, c_{kb})$ is a boolean formula and represented by BDDs, checking satisfiability of $Pre(a_{jb}, c_{kb})$ is cheaper than checking satisfiability of $Pre(a_{ji}, c_{ki})$, which is represented by polyhedra. We should compute

$Pre(a_{ji}, c_{ki})$ which involves the manipulation of the polyhedral representation only if $Pre(a_{jb}, c_{kb})$ is satisfiable. If it is not satisfiable then we will not compute $Pre(a_{ji}, c_{ki})$ since we can deduce that $Pre(a_{jb}, c_{kb}) \wedge Pre(a_{ji}, a_{ki})$ evaluates to *false*. As a result expensive integer manipulation is masked by cheaper boolean manipulation. One consequence of using heuristics like this one, which depend on the discrepancies in the efficiency of manipulating different symbolic representations, is that the `CompSym` class in Fig. 2 becomes dependent on the basic symbolic representation classes `IntSym` and `BoolSym`.

4.2 Subset Check

The subset check algorithm given in Fig. 6 uses the complement operation at the composite formula level (line 11). The worst case time complexity of the complement operation on a composite formula B is exponential in the number of composite atoms in B. For the subset check algorithm, computing the complement of B means that all the composite atoms in B are taken into consideration to decide if a composite atom a is a subset of B. However, deciding if a composite atom a is a subset of a composite formula B does not always require to consider all the composite atoms in B. For instance, let the composite atom a and the composite formula B be,

$$a \equiv (x \wedge y \wedge z \geq 0)$$

and

$$B \equiv (x \wedge z = 0) \vee (x \wedge z \geq 1) \vee (\neg x \wedge z < 0)$$

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 $(x \wedge z = 0)$, $(x \wedge z \geq 1)$, and $(\neg x \wedge z < 0)$, respectively. In order to decide if a is a 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 a subset of $b_1 \vee b_2$) only to conclude that a is subset of B. However, the algorithm given in Fig. 6 will process b_1, b_2, and b_3 by computing the complement of B.

In the light of this observation we propose a more efficient solution to the subset check problem for composite formulas. Given two composite formulas A and B, for each composite atom a in A, our solution iteratively computes the 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 $U \wedge \neg b_k$. 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 a 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 a subset of B. The algorithm is given in Fig. 13. Note that in this algorithm there is no complement operation at the composite formula level. Instead complement is computed for composite atoms in B as needed. The worst case time complexity of the subset check algorithm in Fig. 13 is $O(n_A \times |T|^{n_B} \times \sum_{t \in T}(T^t_{IsEmpty} + T^t_{Intersection}))$. Even though this algorithm also has an exponential worst case time complexity, in the average case we expect it to perform better than the algorithm in Fig. 6.

```
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
```

Fig. 13. Subset check algorithm performing complement at the composite atom level

4.3 Simplification Algorithms

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 complement 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 represented with a single composite atom that represents the formula $((x \vee t) \wedge y > z)$. Since time complexity of manipulating a composite formula is dependent on the number of composite atoms in it, we need to reduce the number

```
1.  Simplify(composite formula A)
2.      a, c, d: composite atom
3.      success: boolean
4.      list_A: list of composite atoms
5.      let list_A be the list of composite atoms in A
6.      a ← head(list_A)
7.      while a ≠ NULL do
8.          c ← next(a)
9.          while c ≠ NULL do
10.             if a ⊆ c then
11.                 remove a from list_A; break
12.             else if c ⊆ a then
13.                 temp ← c
14.                 c ← next(c)
15.                 remove temp from list_A
16.             else
17.                 success ← false
18.                 for each symbolic representation t do
19.                     if a_t ≢ c_t
20.                         if ¬success then
21.                             index ← t ; success ← true
22.                         else success ← false ; break
23.                 if success then
24.                     remove a and c from list_A
25.                     for each symbolic representation t do
26.                         if index = t then
27.                             d_t ← d_t ∨ c_t
28.                         else d_t ← a_t
29.                     insert d to head of list_A
30.                     a ← head(list_A) ; break
31.                 else c ← next(c)
32.         a ← next(a)
```

Fig. 14. Algorithm for simplifying a given composite formula

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 c, can be simplified and represented by a single composite atom d if one of the following holds:

1. a is subset of c. In this case $d \equiv c$.
2. a is superset of c. In this case $d \equiv a$.
3. There exists a symbolic representation t_j s.t. for all symbolic representations t_i s.t. $t_i \neq t_j$, $a_{t_i} \equiv c_{t_i}$. In this case for all t_i s.t. $t_i \neq t_j$, $d_{t_i} \equiv a_{t_i}$ and $d_{t_j} \equiv a_{t_j} \vee b_{t_j}$.

Fig. 14 shows the simplification algorithm. The algorithm takes each pair of composite atoms a and c in a composite formula and checks if union of the formula represented by a and c can be represented by a single composite atom d 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 29 and 30. At line 29 the composite atom d that can replace a and c is inserted to the head of the list of composite atoms and a is set to the head of the list at line 30. This ensures that composite atom d can be compared against all other composite atoms in the list.

We can make the simplification algorithm less aggressive and more efficient in three ways. The first way exploits the fact that the equivalence check on the BDD representation is cheaper than the equivalence check on the polyhedral representation. At line 19 of the *Simplification* algorithm in Fig. 14, a_t and c_t, where t is a symbolic representation, are checked for equivalence. Instead of checking equivalence of a_t and c_t for each symbolic representation t, we can check it for only boolean type. This makes the *Simplify* algorithm less expensive and less aggressive meaning that given two composite atoms a and c, the *Simplify* algorithm will be able to combine a and c into a single composite atom d if a (c) is a subset of c (a) or a_b is equal to c_b, where b denote the BDD part. If a_b is not equal to c_b then the algorithm will conclude that a and c cannot be combined.

The second way is to avoid subset check. At lines 10 and 12 the *Simplification* algorithm in Fig. 14 checks if a is a subset of c or vice versa, respectively. By eliminating subset check between a and c the algorithm will combine two composite formula a and c only if a_b is equal to c_b.

The third way is to consider only a subset of composite atom pairs in a given composite formula. At line 29 of the *Simplify* algorithm in Fig. 14, composite atom d, which is combination of composite atoms a and c, is inserted at the head of $list_A$ and a is set to the head of $list_A$ at line 30. These two steps ensure that there are no pairs of composite atoms in the resulting composite formula A that can be merged. However, if d is inserted at the end of $list_A$ without making a point to the head of $list_A$ then d is not compared with other composite atoms and there may be pairs of composite atoms in the resulting composite formula A which can be merged.

By using the three ways of aggressiveness reduction we obtain 4 different versions of *Simplify* 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:

– $S1$: Number of executions of the inner while loop is $O(n^2)$ and checks equivalence relation only on boolean type and eliminates subset check.

```
1. ComputeEF(composite formula A, composite formula C):
   composite formula
2.     S_new ← A
3.     do
4.         S_old ← S_new
5.         S_new ← Pre(S_old, C) ∨ S_old
6.         S_new ← Simplify(S_new)
7.     while (S_new ⊈ S_old)
8.     return S_new
```

Fig. 15. Algorithm for computing the least fixed point for EF

```
1. PreUnion(A,C, isSubset): composite formula
2.     S_res, A, C: composite formula
3.     isSubset:boolean
4.     isSubset ← true
5.     S_res ← S
6.     for each composite atom a in A do
7.         for each composite atom c in C do
8.             s_k ≡ Pre(a, c)
9.             if s_k ⊈ S
10.                isSubset ← false
11.                S_res ← S_res ∨ s_k
12.    return S_res
```

Fig. 16. A pre-condition algorithm with subset check and union

```
1. EfficientEF(A, C): composite formula
2.     A, C: composite formula
3.     isSubset: boolean
4.     S_new ← A
5.     do
6.         S_old ← S_new
7.         S_new ← PreUnion(S_old, C, isSubset)
8.         S_new ← Simplify(S_new)
9.     while (¬isSubset)
10.    return S_new
```

Fig. 17. A more efficient algorithm for computing the least fixed point for EF

- $S2$: 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.

Note that *Simplify* algorithm in Fig. 14 is not optimal. Consider the composite atoms

$$(\neg x \wedge y \wedge 1 < z \leq 10) \quad (1)$$
$$(x \wedge 2 < z \leq 10) \quad (2)$$
$$(x \wedge y \wedge 2 \leq z \leq 8) \quad (3)$$
$$((\neg x \wedge y) \vee (x \wedge \neg y) \wedge 1 \leq z \leq 2) \quad (4)$$
$$(x \wedge \neg y \wedge 2 < z \leq 8) \quad (5)$$

whose disjunction is equivalent to

$$(x \vee y) \wedge (1 \leq z \leq 10) \quad (6)$$

Simplify algorithm in Fig. 14 cannot discover that disjunction of composite atoms (1), (2), (3), (4), and (5) can be replaced by composite atom (6) since it tries to merge the composite atoms (1), (2), (3), (4), and (5) pairwise.

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

All CTL operators can be defined in terms of least and greatest fixpoints. Temporal operator EF is defined as a least fixpoint as EF $p \equiv \mu x \,.\, p \vee$ EX x. Fig. 15 shows the algorithm for computing the least fixpoint for EF. Given two composite formulas $A \equiv \bigvee_{i=1}^{n_A} a_i$ and $C \equiv \bigvee_{j=1}^{n_C} c_j$, where A represents a set of states and C represents the transition relation, the algorithm computes the set of states that satisfy EF(A) iteratively. S_{new} represents the largest subset of the fixed point computed so far. At line 5 of *ComputeEF* algorithm $S_{new} \equiv \bigvee_{k=1}^{n_S} s_k$ where one of the following holds for s_k:

1. $s_k \equiv Pre(a_i, c_j)$, where $1 \leq i \leq n_A$ and $1 \leq j \leq n_C$, and $s_k \not\subseteq S_{old}$,
2. $s_k \equiv Pre(a_i, c_j)$, where $1 \leq i \leq n_A$ and $1 \leq j \leq n_C$, and $s_k \subseteq S_{old}$,
3. $s_k \subseteq S_{old}$ and there exists no i, j, where $1 \leq i \leq n_A$ and $1 \leq j \leq n_C$, s.t $s_k \equiv Pre(a_i, c_j)$.

Note that composite atoms that satisfy (1) can be used to decide if S_{new} is a subset of S_{old} earlier during the computation of pre-condition and eliminate subset check at line 7 of the algorithm. This may serve as an improvement over the algorithm in Fig. 15 since we can eliminate processing composite atoms in S_{new} that satisfy (3) during the subset check at line 7 of the algorithm. An additional improvement can be achieved by taking the union of s_k with S_{res} only if s_k is not a subset of A and prevent the unnecessary increase in the number of composite atoms in S_{res}. Fig. 16 and 17 show the algorithms *PreUnion*, which computes the pre-condition along with the subset check and the union, and *EfficientEF*, which computes the least fixed point for EF using the *PreUnion* algorithm, respectively.

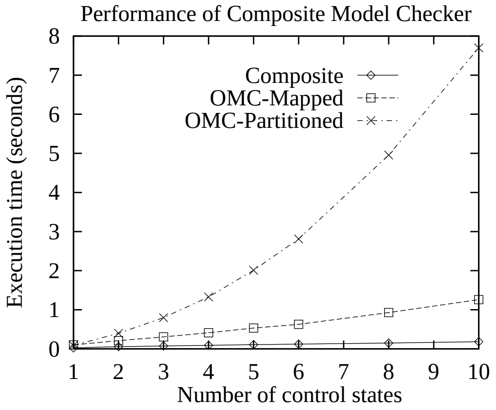

Fig. 19. Performance of the composite model checker and Omega Library model checker (using partitioning or mapping approach) on the bounded-buffer producer-consumer example

5 A Simple Example

In Fig. 18 we show a simple producer-consumer system. Both *producer* and *consumer* components have N control states. Producer produces an item only when it is in control state N and there is available space in the buffer ($count < size$). When it produces an item it increases *produced* and *count* by 1. Similarly, consumer consumes an item only when it is in control state N and there is an item in the buffer ($count > 0$). When it consumes an item it increases *consumed* by 1 and decreases *count* by 1. An invariant of this system is $count \leq size \wedge produced - consumed = count$.

Initial condition for this system can be represented with the composite formula:
$pstate = 1 \wedge cstate = 1 \wedge count = 0 \wedge produced = 0 \wedge consumed = 0 \wedge size >= 0$
where *pstate* and *cstate* are variables introduced to model the control states of the producer and the consumer. The self-loop on state N for the producer can be represented with the composite formula:
$pstate = N \wedge pstate' = N \wedge count < size \wedge produced' = produced + 1 \wedge count' = count + 1$
The overall transition relation is the disjunction of the formulas that correspond to each arc in Fig. 18. (We are assuming that if a variable is not modified it preserves its value. These constraints have to be added to the composite formula before generating a `CompSym` object).

We used this example to compare the performance of our composite model checker with Omega Library Model Checker (OMC) presented in [BGP97,BGP99]. OMC uses the polyhedral representation of arithmetic constraints as a symbolic representation. To represent the control states of the system given in Fig. 18 in such a tool, there are two options, 1) to partition the state space based on the control states, creating N partition classes, 2) to map the control states to an integer variable. Either option is not very efficient because of the high complexity of manipulating arithmetic constraint representations. In the Composite Symbolic Library the control states in the above example are mapped to an enumerated variable which is encoded using BDDs. Integer variables are still encoded using the polyhedral representation, however the unnecessary mapping of the enumerated variables to integers is prevented. Fig. 19 shows the execution time of the composite model checker and the OMC (using both partitioning and integer-mapping) with the increasing number of control states for the system given in Fig. 18. Although this is a small example, it demonstrates the inefficiency of using a model checker which is solely based on polyhedral representations.

6 Experimental Evaluation of the Heuristics

We have experimented with the heuristics explained in Section 4 using a large set of specifications which we describe below. Table 1 shows the different properties verified for each specification. Each instance is labeled using NAME[n_{pc}]-[n_{pr}] where n_{pc} and n_{pr} are the number of processes and the property number, respectively. Specifications of all these examples and properties are available at:
http://www.cs.ucsb.edu/˜bultan/composite/

- BAKERY[2,3,4] and TICKET[2,3,4] are mutual exclusion protocols (for (2,3,4) processes, respectively). We verified both mutual exclusion (BAKERY[2,3,4]-1, TICKET[2,3,4]-1) and starvation-freedom (BAKERY[2,3,4]-2, TICKET[2,3,4]-2) properties for these protocols.
- We verified three properties for sleeping barber monitor specifications with 2, 3, and 4 customer processes and one barber process (BARBER[2,3,4]-[1,2,3]). We also verified the three properties (BARBERP-[1,2,3]) on the parameterized system.
- We analyzed a parameterized cache coherence protocol specification given in [DB01]. We verified all the properties given in [DB01].
- INSERTIONSORT is a specification from [DP01] for array bound checking of an implementation of insertion sort algorithm.
- PC[5,10,30] is a bounded buffer producer consumer system as given in Section 5 with 5, 10, or 30 control states.
- RW[16,32,64] is a monitor specification for the readers-writers problem for various number of processes [And91].
- LIGHTCONTROL and SIS are two reactive software specifications. LIGHTCONTROL is an office light control system specification written in statecharts [BYK01]. SIS is specification of a safety injection system for a nuclear reactor [CP93]

We obtained the experimental results on a SUN ULTRA 10 workstation with 768 Mbytes of memory, running SunOs 5.7.

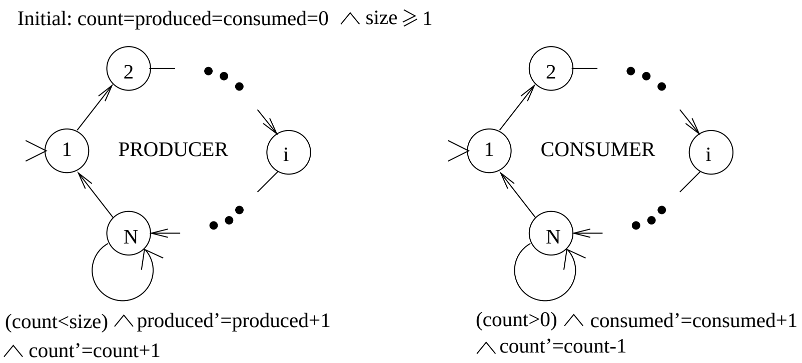

Fig. 18. A simple bounded-buffer producer-consumer example

Problem Instance	Property
BAKERY2-1	$AG(\neg(p1 = cs \wedge p2 = cs))$
BAKERY3-1	$AG(\neg((p1 = cs \wedge p2 = cs) \vee (p1 = cs \wedge p3 = cs) \vee (p2 = cs \wedge p3 = cs)))$
BAKERY[2,3]-2	$AG(\neg(p1 = try) \vee AF(p1 = cs))$
TICKET2-1	$AG(\neg(p1 = cs \wedge p2 = cs))$
TICKET3-1	$AG(\neg(p1 = cs \wedge p2 = cs \vee p1 = cs \wedge p3 = cs \vee p2 = cs \wedge p3 = cs))$
TICKET[2,3]-2	$AG(\neg(p1 = try) \vee AF(p1 = cs))$
BARBER[2,3,4,P]-1	$AG(chair \leq 1)$
BARBER[2,3,4,P]-2	$AG(open \leq 1)$
BARBER[2,3,4,P]-3	$AG(barber \leq 1)$
COHERENCE-1	$AG(\neg((xShared \geq 1 \wedge xExclusive \geq 1) \vee xExclusive \geq 2))$
COHERENCE-2	$AG(xWaitS \geq 1 \Rightarrow AF(xShared \geq 1))$
COHERENCE-3	$AG(xWaitS \geq n \Rightarrow AF(xShared \geq n))$
COHERENCE-4	$AG(xWaitE \geq 1 \Rightarrow AF(xExclusive \geq 1))$
COHERENCE-REF-1	$AG(\neg((xShared >= 1 \wedge xExclusive >= 1) \vee xExclusive >= 2))$
COHERENCE-REF-2	$AG(xWaitS \geq 1 \Rightarrow AF(xShared \geq 1))$
COHERENCE-REF-3	$AG(xWaitS \geq n \Rightarrow AF(xShared \geq n))$
INSERTIONSORT	$AG(\neg((pc = entryA1 \wedge k \geq n) \vee (pc = entryA1 \wedge k \leq -1) \vee (pc = entryA3 \wedge i \geq n-1) \vee$ $(pc = entryA3 \wedge i \leq -2) \vee (pc = entryA2 \wedge i \leq -1) \vee (pc = entryA2 \wedge i \leq -2) \vee$ $(pc = entryA2 \wedge i \geq n-1) \vee (pc = entryA2 \wedge i \geq n-1)))$
PC[5,10,30]	$AG(produced - consumed = count \wedge count <= size)$
RW[16,32,64,P]	$AG(busy \Rightarrow nr = 0)$
SIS-1	$AG(Inject \Rightarrow Pressure = TooLow)$
SIS-2	$AG((Reset \wedge \neg(Pressure = TooHigh) \Rightarrow \neg Overridden) \wedge (Reset \wedge Pressure = TooLow \Rightarrow Inject))$
LIGHTCONTROL	$AG(count > 1 \iff Office = Occupied \wedge Occupants = Multiple)$

Table 1. List of problem instances used in the experiments

In Tables 2 and 3 we show the sizes of the problem instances we used in our experiments. Each row in Table 2 shows the size of the composite symbolic representation for the transition relation used in that instance. The size of a composite representation is shown in terms of the number of composite atoms, the number of polyhedra, the number of equality (EQ) and greater-than-or-equal-to (GEQ) constraints, the number of BDD nodes, the number of integer variables, and the number of boolean variables in it. Table 3 shows the size of the maximum fixpoint iterate for the optimized version of the Composite Symbolic Library with all the presented heuristics included. For most of the problem instances verification procedure runs out of memory if the heuristics are not used.

Experimental results for the verifier with different versions of the simplification algorithm and without simplification is given in Table 4. 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. Without simplification, for most of the examples, the verifier ran out of memory.

The results in Table 5 show that combining the subset check and the union computations with the pre-condition computation speeds up 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

Problem Instance	Transition Relation Size					
	Composite	Polyhedra	EQ, GEQ	BDD	# integer	# boolean
BAKERY2-[1,2]	6	8	32	69	2	4
BAKERY3-[1,2]	9	121	126	165	3	6
TICKET2-[1,2]	6	6	38	69	4	4
TICKET3-[1,2]	9	9	66	165	5	6
BARBER2-[1,2,3]	8	8	48	88	3	4
BARBER3-[1,2,3]	10	10	60	140	3	5
BARBER4-[1,2,3]	12	12	72	204	3	6
BARBERP-[1,2,3]	6	6	62	32	6	2
COHERENCE-[1,2,3,4]	10	10	120	94	6	4
COHERENCE-REF-[1,2,3]	10	10	120	88	6	4
INSERTIONSORT	8	8	27	56	3	3
PC5	12	12	62	222	4	6
PC10	22	22	112	517	4	8
PC30	62	62	312	1891	4	10
RW16	32	32	64	1570	1	17
RW32	64	64	128	6210	1	33
RW64	128	128	256	24706	1	65
RWP	4	4	56	11	7	1
SIS-1	8	10	50	117	3	6
SIS-2	8	14	1573	117	6	6
LIGHTCONTROL	12	12	25	271	1	7

Table 2. Sizes of the transition relations for the problem instances used in the experiments

smaller size. 2) Only the disjuncts which are result 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 this heuristic is %29.

Verification times with and without masking the integer pre-condition computation by boolean satisfiability check are given in Table 5. Results show that masking the 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 the liveness properties involve two fixpoint computations: one for EG and one for EF[2]. 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 the intersection causes a quadratic increase (whereas the union causes a linear increase) in the composite formula size, EG fixpoint iterates are likely to grow faster. Average speedup for masking heuristic is %12.

Verification times for the two different versions of the subset check algorithm are given in Table 5. In most of the experiments the efficient subset check algorithm shown in Fig. 13 performs better than the subset check algorithm shown in Fig. 6 which computes the complement operation at the composite formula level. The results demonstrate that processing composite atoms as needed and performing the complement operation at the composite atom level instead of composite formula level is more efficient. Average speedup for efficient subset check heuristic is %10.

7 Omega Library vs. Composite Symbolic Library

In Section 5 we have shown that the composite model checker performs superior to the OMC-Partitioned for the example given in Figure 18. It is clear from Figure 19 that partitioning the state space does not scale with the increasing number of control states. In this Section, we compare the composite representation against the integer-mapping approach (used in OMC-Mapped in Section 5) using a larger set of examples to further evaluate the power of the composite symbolic model checking approach. In the integer-mapping approach enumerated and boolean variables are mapped to integer variables and verification is performed using only the polyhedral representation of the Presburger arithmetic formulas provided by the Omega Library.

Table 6 shows the verification time and the memory usage for the integer-mapping and the composite representation approaches. The results show that the composite representation approach scales better than the integer-mapping approach. The performance of the integer-mapping approach degrades relative to the performance of the composite representation approach with the increasing number of boolean variables. The results also show that for most of the problem instances which can be verified by the composite representation approach using approximate fixpoint computations, the analysis

[2] 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 $I \wedge EF(\neg p)$

Problem Instance	Maximum Fixpoint Iterate Size				Fixpoints	
	Composite	Polyhedra	EQ, GEQ	BDD	EF	EG
BAKERY2-1	6	10	20	28	4	–
BAKERY2-2	4	5	10	21	1	9
BAKERY3-1	22	61	183	171	5	–
BAKERY3-2	16	48	146	141	4	15
TICKET2-1	6	11	36	31	9	–
TICKET2-2	8	25	74	41	7	5
TICKET3-1	19	28	117	168	11	–
TICKET3-2	27	54	191	195	6	8
BARBER2-1	14	29	87	60	9	–
BARBER2-2	4	4	12	13	5	–
BARBER2-3	9	10	30	42	11	–
BARBER3-1	16	35	105	90	10	–
BARBER3-2	4	4	12	15	5	–
BARBER3-3	12	14	42	75	12	–
BARBER4-1	18	41	123	128	11	–
BARBER4-2	4	4	12	17	5	–
BARBER4-3	15	18	54	122	13	–
BARBERP-1	4	24	167	10	8	–
BARBERP-2	4	6	26	10	5	–
BARBERP-3	5	14	73	13	11	–
COHERENCE-1	5	7	42	23	4	–
COHERENCE-2	11	43	258	51	5	3
COHERENCE-3	15	81	494	72	8	3
COHERENCE-4	15	20	111	73	13	11
COHERENCE-REF-1	4	6	36	20	4	–
COHERENCE-REF-2	7	8	46	33	5	3
COHERENCE-REF-3	11	186	1116	53	9	3
INSERTIONSORT	5	8	10	19	5	–
PC5	1	3	8	7	1	–
PC10	1	3	8	7	1	–
PC30	1	3	8	9	1	–
RW16	1	1	1	2	1	–
RW32	1	1	1	2	1	–
RW64	1	1	1	2	1	–
RWP	1	1	7	2	1	–
SIS-1	1	1	1	3	1	–
SIS-2	2	7	49	14	2	–
LIGHTCONTROL	4	4	5	35	3	–

Table 3. Maximum fixpoint iterate sizes and the number of fixpoint iterations for the optimized version of the Composite Symbolic Library for problem instances used in the experiments

results for the integer-mapping approach are inconclusive (marked with "*" in Table 6) although we used the same approximation technique [BGP99] in both cases.

8 Conclusions

In this paper, we have presented the Composite Symbolic Library, a symbolic manipulator for model-checking systems with heterogeneous data types. The Composite Symbolic Library combines different symbolic representations under a common interface. It can be used as a platform to integrate different symbolic representations. Using composite representations one can improve the efficiency of verification procedures by mapping each variable type in the input specification to a suitable symbolic representation.

The `Symbolic` interface provided by our tool can be useful in integrating different symbolic libraries. Once a wrapper for a symbolic library is written the internal representation of that library will be hidden. This can also help in comparing performances of different symbolic representations by isolating them from the verification procedures.

Using the Composite Symbolic Library we were able to develop polymorphic verification procedures which are oblivious to the symbolic representation used. Hence, the decision of which symbolic representation to use can be made at run-time, based on the input specification. If the input specification has only boolean and enumerated variables, then our verifier becomes a BDD-based symbolic model checker. However, if both integer and boolean variables are present in the input specification, then it is able to use arithmetic constraints and BDDs together using the composite representation.

We have also improved performance of the Composite Symbolic Library by developing heuristics for efficient manipulation of the composite symbolic representation.

Problem	S2-S3-S4		S1		S2		S3		S4		None	
Instance	T	M	T	M	T	M	T	M	T	M	T	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	7.9	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	6.76	24	5.17	21.1	5.64	24	6.53	24	7.8	24	173	228
BARBERP-2	0.22	9.4	0.13	9.3	0.16	9.3	0.19	9.3	0.20	9.3	0.38	10.3
BARBERP-3	1.17	13.5	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	24.6	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	0.48	11.6	>139	>181	>139	>181	>139	>181	>139	>181	↑	↑
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 4. Verification time (T, in seconds) and memory (M, in Mbytes) results for different versions of the simplification heuristic vs. no simplification (↑ means that the program ran out of memory, $> x$ means that the execution did not terminate in x seconds, and (L) indicates that the specification has been verified for a liveness property)

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 of memory. Heuristics for combining the pre-condition computation with the subset check and the union computations, avoiding the complement operation at the composite formula level and performing it at the composite atom level, and masking the expensive integer manipulation operations with the boolean manipulation operations also improve the verification times significantly.

We would like to enrich the Composite Symbolic Library with new symbolic representations. Some candidates are automata representation for linear arithmetic formulas and shape graphs for heap variables. Considering the object-oriented design of the Composite Symbolic Library, we believe that integration of these new symbolic representations will be feasible.

Preliminary results from this paper were presented in [YKTB01] and [YKB02].

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.

[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.

Problem Instance	−Subset T	−Subset M	−PreUnion T	−PreUnion M	−Mask T	−Mask M	All-Heuristics T	All-Heuristics M
BAKERY2-1	0.26	10.9	0.32	7.8	0.24	8.7	0.21	7.8
(L)BAKERY2-2	0.34	9.4	0.3	7.9	0.41	8	0.26	7.9
BAKERY3-1	20.99	268	20.37	16.1	8.34	28.8	8.26	19.6
(L)BAKERY3-2	54	69	50.04	34.7	57.34	36.6	51.32	34.7
TICKET2-1	1.16	18.3	1.22	9.6	1.14	13.2	1.07	10.2
(L)TICKET2-2	3.31	23.2	3.26	12.3	3.35	19.7	3.13	13.8
TICKET3-1	18.87	159	29.14	20.1	14.76	37.4	14.71	28
(L)TICKET3-2	36.47	204	66.53	34.2	32.67	49.9	29.73	29
BARBER2-1	8.03	122	6.32	15.7	4.65	21.4	4.62	17.7
BARBER2-2	0.2	10.1	0.37	8.3	0.24	9	0.27	8.8
BARBER2-3	1.69	34.4	2	11.1	1.63	14.5	1.58	12.8
BARBER3-1	21.75	299	11.53	18.8	11.07	31.1	10.93	26.6
BARBER3-2	0.21	11.2	0.46	8.6	0.36	9.7	0.35	9.5
BARBER3-3	6.03	105	4.15	13.7	4.12	20.5	4.12	18.1
BARBER4-1	43.77	554	20.62	22.8	22.21	44.4	21.98	38.5
BARBER4-2	0.27	12.3	0.59	8.9	0.44	10.4	0.43	10.1
BARBER4-3	16.94	264	8.38	16.7	8.87	29.1	8.76	25.9
BARBERP-1	4.64	31.9	6.48	18.8	6.83	25.3	6.76	24
BARBERP-2	0.25	10.4	0.44	9.4	0.24	9.2	0.22	9.4
BARBERP-3	1.59	23.5	5.11	21.8	1.27	14	1.17	13.5
COHERENCE-1	0.37	13.4	0.86	10	0.38	13	0.34	11.3
(L)COHERENCE-2	4.92	53.9	4.88	15.4	3.43	29.3	2.74	14.8
(L)COHERENCE-3	21.61	169	26.45	35.4	13.93	91.1	11.97	29.8
(L)COHERENCE-4	15.65	96.2	13.38	22.1	25.64	49.7	13.14	24.6
COHERENCE-REF-1	0.28	11.9	0.56	10.1	0.33	12.7	0.27	11
(L)COHERENCE-REF-2	1.17	21.2	1.93	11.2	1.26	14.3	0.97	11.4
(L)COHERENCE-REF-3	0.59	18.2	0.50	11.6	1.01	11.8	0.48	11.6
INSERTIONSORT	0.2	9.1	0.26	8.6	0.3	10.8	0.2	8.5
PC5	0.07	7.6	0.12	7.7	0.07	7.8	0.07	7.7
PC10	0.09	8.4	0.22	8.5	0.09	8.6	0.09	8.5
PC30	0.25	11.6	0.62	11.7	0.26	11.8	0.25	11.8
RW16	0.02	8.1	0.03	8.1	0.02	8.1	0.02	8.1
RW32	0.03	10.8	0.05	10.8	0.04	10.8	0.03	10.8
RW64	0.04	20.6	0.1	20.6	0.08	20.6	0.05	20.6
RWP	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
SIS-2	0.07	19.4	0.08	19.4	0.38	27.2	0.07	19.4
LIGHTCONTROL	0.13	8.8	0.17	7.6	0.12	8.5	0.12	7.9

Table 5. Verification time (T, in seconds) and memory (M, in Mbytes) results demonstrating the impact of different heuristics (− denotes exclusion of the specified heuristic and All-Heuristics denotes that all the heuristics are enabled)

[BGL98] T. Bultan, R. Gerber, and C. League. Verifying systems with integer constraints and boolean predicates: A composite approach. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 113–123, March 1998.

[BGL+00a] S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueb, J. Rushby, V. Rusu, H. Saidi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In *Proceedings of the Fifth Langley Formal Methods Workshop*, June 2000.

[BGL00b] 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.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, June 1997.

[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.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of the 10th International Conference for Computer-Aided Verification (CAV'98)*, 1998.

[BPRue] T. Ball, A. Podelski, and S.K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In *Journal of Software Tools for Technology Transfer*, 2002.This issue.

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

[BS00] R. Bharadwaj and S. Sims. Salsa: Combining constraint solvers with BDDs for au-

Problem Instance	Integer-Mapping T	Integer-Mapping M	Composite Representation T	Composite Representation M
BAKERY2-1	0.18	3	0.21	7.8
(L)BAKERY2-2	0.27	3	0.26	7.9
BAKERY3-1	22	18	8.26	19.6
(L)BAKERY3-2	61.2	45.2	51.32	34.7
TICKET2-1	*	*	1.07	10.2
(L)TICKET2-2	*	*	3.13	13.8
TICKET3-1	*	*	14.71	28
(L)TICKET3-2	*	*	29.73	29
BARBER2-1	7	9.5	4.62	17.7
BARBER2-2	0.21	5.4	0.27	8.8
BARBER2-3	1.88	6.5	1.58	12.8
BARBER3-1	38.4	18.7	10.93	26.6
BARBER3-2	0.4	7.4	0.35	9.5
BARBER3-3	12.7	12.4	4.12	18.1
BARBER4-1	166.9	36	21.98	38.5
BARBER4-2	0.9	10	0.43	10.1
BARBER4-3	84	25.6	8.76	25.9
BARBERP-1	3.31	6.4	6.76	24
BARBERP-2	0.14	3.9	0.22	9.4
BARBERP-3	0.86	4.8	1.17	13.5
COHERENCE-1	0.6	13.6	0.34	11.3
(L)COHERENCE-2	8	20.6	2.74	14.8
(L)COHERENCE-3	22.2	34.7	11.97	29.8
(L)COHERENCE-4	*	*	13.14	24.6
COHERENCE-REF-1	0.4	13.3	0.27	11
(L)COHERENCE-REF-2	*	*	0.97	11.4
(L)COHERENCE-REF-3	*	*	0.48	11.6
INSERTIONSORT	0.2	3	0.2	8.5
PC5	0.09	3.3	0.07	7.7
PC10	0.22	4.8	0.09	8.5
PC30	1.3	10.9	0.25	11.8
RW16	0.07	26	0.02	8.1
RW32	↑	↑	0.03	10.8
RW64	↑	↑	0.05	20.6
RWP	0.01	3.2	0.01	9
SIS-1	0.02	6.6	0.01	7.5
SIS-2	0.8	29.4	0.07	19.4
LIGHTCONTROL	7.13	1385.4	0.12	7.9

Table 6. Verification time (T, in seconds) and memory (M, in Mbytes) results for the integer-mapping and the composite representation (↑ means that the program ran out of memory, * means that the analysis was not precise enough to verify the property and (L) indicates that the specification has been verified for a liveness property)

tomatic invariant checking. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 378–394. Springer, April 2000.

[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.

[CABN97] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Combining constraint solving and symbolic model checking for a class of systems with non-linear constraints. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 316–327. Springer, June 1997.

[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/.

[DB01] 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)*, 2001.

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

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, 1994.

[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.

[KMP+95] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library interface guide. Technical Report CS-TR-3445, Department of Computer Science, University of Maryland, College Park, March 1995.

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

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

[Sai00] H. Saidi. Model checking guided abstraction and analysis. In *Proceedings of Statica Analysis Symposium*, Lecture Notes in Computer Science. Springer, 2000.

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

[YKB02] T. Yavuz-Kahveci and T. Bultan. Heuristics for efficient manipulation of composite constraints. In *Proceedings of the 4th International Workshop on Frontiers of Combining Systems (FroCoS 2002)*, 2002.

[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.