A Library for Composite Symbolic Representations

Tuba Yavuz-Kahveci, Murat Tuncer, and Tevfik Bultan

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

Abstract. In this paper, we present the design and the implementation of a composite model checking library. Our tool combines different symbolic representations, such as BDDs for representing boolean logic formulas and polyhedral representations for linear arithmetic formulas, with a single interface. Based on this common interface, these data structures are combined using what we call a *composite representation*. We used an object-oriented design to implement the composite symbolic library. We imported CUDD (a BDD library) and 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 forms 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

In symbolic model checking sets of states and transitions are represented symbolically (implicitly) to avoid the state-space explosion problem [BCM$^+$90,McM93]. Success of symbolic model checking has been mainly due to efficiency of the data structures used to represent the state space. For example, binary decision diagrams (BDDs) [Bry86] have been successfully used in verification of finite-state systems which could not be verified explicitly due to size of the state space [BCM$^+$90,McM93]. Linear arithmetic constraint representations have been used in verification of real-time systems, and infinite-state systems [ACH$^+$95,AHH96,BGP99,HRP94] which are not possible to verify using explicit representations. Any data structure that supports operations such as intersection, union, complement, equivalence checking and existential quantifier elimination (used to implement relational image computations) can be used as a

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

symbolic representation in model checking. The motivation is to find symbolic representations which can represent the state space compactly to avoid state-space explosion problem. However, symbolic representations may have their own deficiencies. For example BDDs are incapable of representing infinite sets. On the other hand linear arithmetic constraint representations, which are capable of representing infinite sets, are expensive to manipulate due to increased expressivity.

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 [Sai00]. 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 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 get 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 several 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.

We use an object oriented design for our tool. First we declare an interface for symbolic representations. This interface is specified as an abstract class. All symbolic representations are defined as classes derived from this interface. We integrated CUDD and Omega Library to our tool by writing wrappers around them which implements this interface. This makes it possible for our verifier to interact with these libraries using a single interface. The symbolic representations based on these tools form the basic representation types of our composite library. Our composite class is also derived from the abstract symbolic representation class. A composite representation consists of a disjunction of composite atoms where each composite atom is a conjunction of basic symbolic representations. Composite class manipulates this representation to compute operations such as union, intersection, complement, forward-image, backward-image, equivalence check, etc.

There have been other studies which use different symbolic representations together. In [CABN97], Chan *et al.* present a technique in which (both linear and non-linear) constraints are mapped to BDD variables (similar representations were also used in [AB96,AG93]) 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 modular design of our tool it should be easy to extend it with automata-based linear constraint representations.

Symbolic Analysis Laboratory (SAL) is a recent attempt to develop a framework for combining different tools in verifying properties of concurrent systems [BGL$^+$00a]. The heart of the tool is a language for specifying concurrent systems in a compositional manner. Our composite symbolic library is a low-level approach compared to SAL. We are combining different libraries at the symbolic representation level as opposed to developing a specification language to integrate different tools.

The rest of the paper is organized as follows. We explain the design of our composite symbolic library in Section 2. In Section 3, we describe the algorithms for manipulating composite representations. Section 4 presents the polymorphic verification procedure. In Section 5 we show the performance of the composite model checker on a simple example. Finally, in Section 6 we conclude and give some future directions.

2 Composite Symbolic Library

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 BDD representation, and integers can be mapped to an arithmetic constraint representation. Then, 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. We encode the sets of system states as a disjunction of conjunctively partitioned type specific representations (e.g., 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). The forward and backward image computations are computed independently for each symbolic representation by exploiting the conjunctive partitioning of the atomic events. We also implement algorithms for intersection, union, complement and equivalence checking computations for the disjunctive *composite* representation that use the corresponding methods for different symbolic representations. The key observation here is the fact that conjunctive partitioning of the atomic events allows forward and backward image computations to distribute over different symbolic representations.

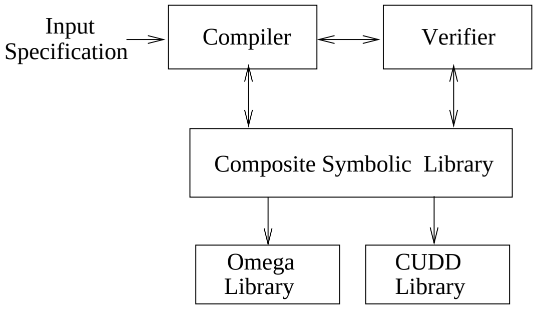

Fig. 1. Architecture of the composite model checker

Our current implementation of the composite symbolic library uses two symbolic representations: BDDs and polyhedral representation for Presburger arithmetic formulas. 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 a general picture of our composite model checking system. We will focus on the symbolic library and verifier parts of the system in this paper.

We implemented our 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. Class `BoolSym` serves as a wrapper for

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

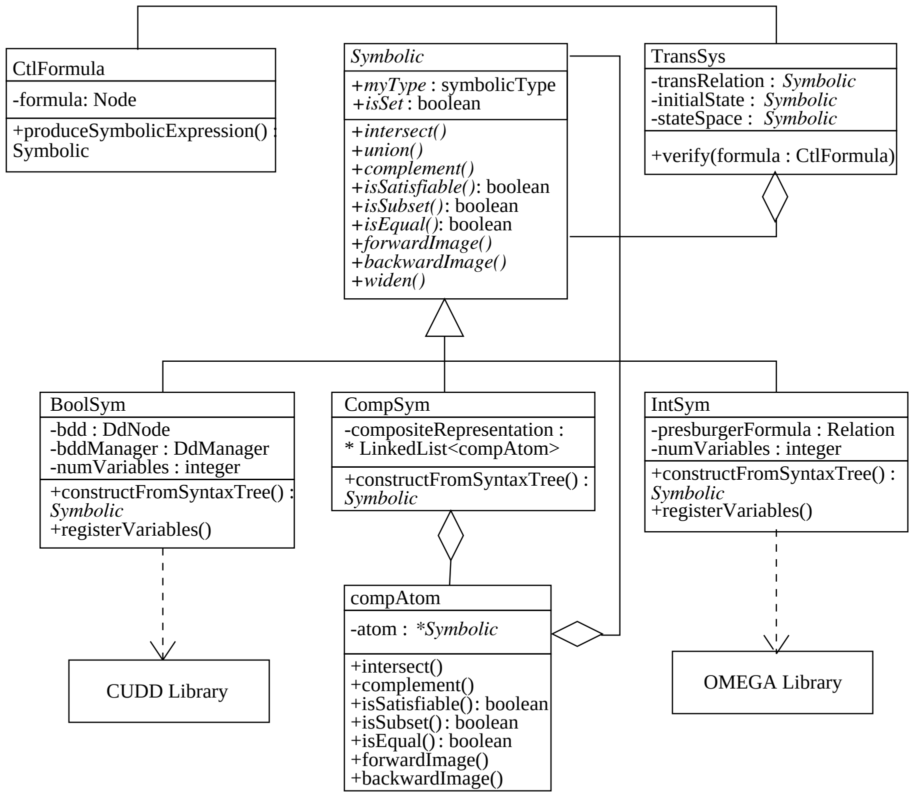

Fig. 2. Class diagram for the composite symbolic library

the BDD library CUDD [CUD]. It is derived from the abstract class `Symbolic`. Similarly, `IntSym` is also derived from abstract class `Symbolic` and serves as a wrapper for the Omega Library [Ome].

The class `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. Note that this design is an instance of the composite design pattern given in [GHJV94].

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*. The syntax of a composite formula is defined as follows:

$$CF ::= CF \wedge CF \mid CF \vee CF \mid \neg CF \mid BF \mid IF$$
$$BF ::= BF \wedge BF \mid BF \vee BF \mid \neg BF \mid Term_{bool}$$
$$IF ::= IF \wedge IF \mid IF \vee IF \mid \neg\, IF \mid Term_{int}\ Rop\ Term_{int}$$
$$Term_{bool} ::= id_{bool} \mid true \mid false$$
$$Term_{int} ::= Term_{int}\ Aop\ Term_{int} \mid -Term_{int} \mid id_{int} \mid constant$$

where CF, BF, IF, *Rop*, and *Aop* denote composite formula, boolean formula, integer formula, relational operator, and arithmetic operator, respectively. Since symbolic representations in our composite library currently support only boolean and linear arithmetic formulas, we restrict arithmetic operators to $+$ and $-$ (we actually 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 current state variables and primed variables to denote next state variables. A method called `registerVariables` in `BoolSym` and `IntSym` is used to register current and next state variable names during the initialization of the representation.

Given a composite formula, the method `constructFromSyntaxTree()` in `CompSym` traverses the syntax tree and calls `constructFromSyntaxTree()` method of `BoolSym` when a boolean formula is encountered and calls `constructFromSyntaxTree()` method of `IntSym` when an integer formula is encountered. In `CompSym`, a composite formula, A, is represented in Disjunctive Normal Form (DNF) as

$$A = \bigvee_{i=1}^{n} \bigwedge_{j=1}^{t} a_{ij}$$

where a_{ij} denotes the the formula of type j in the ith disjunct, and n and t denote the number of disjuncts and the number of types, respectively.

Each disjunct $\wedge_{j=1}^{t} a_{ij}$ 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 DNF form above. Figure 3 shows internal representation of the composite formula

$$(a > 0 \ \wedge \ a' = a + 1 \ \wedge \ b') \ \vee \ (a \leq 0 \ \wedge \ a' = a \ \wedge \ b' = b)$$

in a `CompSym` object. The field *atom* is an array of pointer to class `Symbolic` and the size of the array is the number of basic symbolic representations.

`CompSym` and `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 Omega Library.

A `Simplifier` class implements a simplifier engine that reduces the number of disjuncts in the composite representations. Given a disjunctive formula A it searches for pairs of disjuncts $\wedge_{j=1}^{t} a_{ij}$ and $\wedge_{j=1}^{t} a_{kj}$ that can be expressed as a single disjunct $\wedge_{j=1}^{t} b_j$. Two disjuncts $\wedge_{j=1}^{t} a_{ij}$ and $\wedge_{j=1}^{t} a_{kj}$ can be simplified to a single disjunct $\wedge_{j=1}^{t} b_j$ if one of the following holds:

- $\wedge_{j=1}^{t} a_{ij}$ is subset of $\wedge_{j=1}^{t} a_{kj}$. Then $\wedge_{j=1}^{t} b_j = \wedge_{j=1}^{t} a_{kj}$.

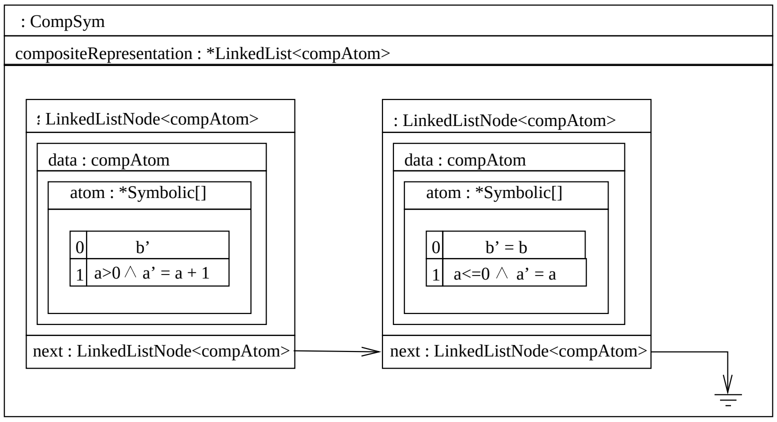

Fig. 3. An instance of `CompSym` class

- $\wedge_{j=1}^{t} a_{ij}$ is superset of $\wedge_{j=1}^{t} a_{kj}$. Then $\wedge_{j=1}^{t} b_j = \wedge_{j=1}^{t} a_{ij}$.
- There exists j such that a_{ij} is not equal to a_{kj} and for $1 \leq m \leq t$, $m \neq j$, a_{im} is equal to a_{km}. Then for $1 \leq m \leq t, m \neq j$, $b_m = a_{im}$ and $b_j = a_{ij} \vee a_{kj}$.

3 Algorithms for Manipulating Composite Representations

In this section, we present the algorithms used in `compAtom` and `CompSym` classes to implement the methods of `Symbolic` interface such as intersection, union, complement, image computations, subset, equality and satisfiability checks. Note that the algorithms given below are independent of the type and number of basic symbolic representations used.

Throughout this section, `CompSym` objects A and B are assumed to be in the following forms:

$$A = \bigvee_{i=1}^{n_A} \bigwedge_{j=1}^{t} a_{ij} \quad \text{and} \quad B = \bigvee_{i=1}^{n_B} \bigwedge_{j=1}^{t} b_{ij}$$

and $n_A(n_B)$, t, and T^i_{Op} denote the number of `compAtom` objects in `CompSym` object $A(B)$, the number of basic symbolic representations in the composite library, and time complexity of i^{th} symbolic representation for operation Op.

Subset Relation Checking: Given two `compAtom` objects a and b, a.`isSubset`(b) is evaluated by checking the corresponding symbolic representations in a and b for subset relation (step 2 of the algorithm given below). Checking subset relation for `CompSym` objects is more complicated. Given two `CompSym` objects, A and B, A.`isSubset`(B) is evaluated as shown in the algorithm below. First, both A and B are simplified, which has a time complexity of $O((n_A^3 + n_B^3) \times \sum_{i=1}^{t}(T^i_{equal} + T^i_{isSubset}))$. Then for each `compAtom` object a in A we check if a is a subset of B.

An efficient way to check if a `compAtom` object a is subset of `CompSym` object B is to compare a with each `compAtom` object b in B till a.`isSubset`(b) evaluates to true. This is done in steps 7-11 below. However, if no such b can be found, this does not mean that a is not a subset of B. Next, we create a new `CompSym` object C, which consists of a single `compAtom` object a. We take the intersection of C and *not* B to obtain the `CompSym` object D. Then D is checked for satisfiability. If D is satisfiable then it means a is not a subset of B (steps 13-19). Time complexity of checking subset relation between two `CompSym` objects, A and B, is $O(n_A \times n_B \times t^{2n_B} \times \sum_{i=1}^{t} T^i_{isSatisfiable})$.

```
boolean compAtom::isSubset(compAtom other)
 for i=1 to numBasicTypes do
     if not atom[i].isSubset(other.getAtom(basicTypes[i])) then
        return false;
return true;
```

```
boolean CompSym::isSubset(Symbolic other)
1  compAtom thisatom,otheratom; boolean found;
2  LinkedList<compAtom> otherlist = other.getCompAtomList();
3  this.simplify();
4  other.simplify();
5  for compRep.hasMore() do
6      thisatom = compRep.getNext();
       found = false;
7      for otherlist.hasMore() do
8          otheratom = otherlist.getNext();
9          if thisatom.isSubset(otheratom) then
10            found = true;
11            break;
12     if not found then
13        CompSym newsym1 = new CompSym(thisatom,isSet);
14        CompSym newsym2 = new CompSym(otherlist,isSet);
15        newsym2.complement();
16        newsym1.intersect(newsym2);
17        if newsym1.isSatisfiable() then
18           return false;
19        else break;
20  return true;
```

Equivalence Checking: Checking equivalence of two `compAtom` objects is performed by calling `isEqual()` method of each symbolic representation similar to subset checking. Equivalence of two `CompSym` objects is checked by calling `isSubset()` method of `CompSym` class and time complexity of `isEqual()` method is the same as `CompSym::isSubset()` method.

Satisfiability Checking: Checking satisfiability of a `compAtom` object is performed by calling `isSatisfiable()` method of each symbolic representation. The condition for satisfiability of a `compAtom` object, a, is that each symbolic representation in a must be satisfiable. Satisfiability of a `CompSym` object A

is equivalent to existence of a `compAtom` object a in A such that a is satisfiable. Time complexity of checking satisfiability of `CompSym` object A is $O(n_A \times \sum_{i=1}^{t} T^i_{isSatisfiable})$.

Backward Image Computation: Backward image computation takes the transition relation as the input parameter. Backward image of a `compAtom` object is computed by calling `backwardImage()` method of each symbolic representation and passing the corresponding symbolic representation of the input `compAtom` object, as the parameter. While computing backward image for a `CompSym` object A, a new list of `compAtoms` is created and for each `compAtom` object in A as many copies as the number of `compAtom` objects in the input `CompSym` object are created. On each copy `backwardImage()` method is called with a `compAtom` object in the input `CompSym` object and the resulting `compAtom` object is inserted in the new list. At the end the `compAtom` list of A is replaced with this new list. Time complexity of computing backward image of `CompSym` object A over `CompSym` object B is $O(n_A \times n_B \times \sum_{i=1}^{t} T^i_{backwardImage})$.

```
compAtom::backwardImage(compAtom other)
 for i=1 to numBasicTypes do
     atom[i].backwardImage(other[i]);

CompSym::backwardImage(Symbolic other)
 compAtom thisatom,newatom;
 LinkedList<compAtom> newlist();
 LinkedList<compAtom> otherlist = other.getCompAtomList();
 for compRep.hasMore() do
     thisatom = compRep.getNext();
     for otherlist.hasMore() do
         newatom = thisatom;
         newatom.backwardImage(otherlist.getNext());
         newlist.insert(newatom);
 compRep = newlist;
```

Intersection : Given two composite formula, A and B, we define A intersection B as

$$A \wedge B = \bigvee_{i=1}^{n_A} \bigvee_{k=1}^{n_B} \bigwedge_{j=1}^{t} (a_{ij} \wedge b_{kj}) \tag{1}$$

Intersection of two `compAtom` objects is computed by calling `intersect()` method of each symbolic representation and passing the corresponding symbolic representation in the input `compAtom` object. To compute intersection of two `CompSym` objects, A and B, a new list of `compAtoms` is created and for each `compAtom` a in A and for each `compAtom` object b in B, intersection of a and b is computed and the resulting `compAtom` object is inserted into the new list. At the end `compAtom` list of A is replaced with the new list. The number of disjuncts in the resulting `CompSym` object after intersection of two `CompSym` objects, A and B, is $O(n_A \times n_B)$.

Complement : Given a composite formula A we define A's complement as

$$\neg A = \bigvee_{1 \le k \le t} \bigwedge_{i=1}^{n_A} \neg a_{ik} \tag{2}$$

Complement of a `compAtom` object is computed by creating a new `Compsym` object for negation of each symbolic representation in the `compAtom` object. Then the union of each newly created `CompSym` object is the result of the complement as seen in the algorithm below. To compute the complement of a `CompSym` object A, a new `CompSym` object B, which is initialized to *True*, is created. For each `compAtom` object a in A, complement of a is intersected with B (steps 4-6). The number of disjuncts in the resulting `CompSym` object after complementation of `CompSym` object A is $O(t^{n_A})$.

```
CompSym compAtom::complement()
CompSym result,temp;
Symbolic sym;
result = null;
for i=1 to numBasicTypes do
    if result != null then
       sym = atom[i];
       sym.complement();
       result.union(new CompSym(sym,isSet));
    else
       result = new CompSym(atom[i],isSet);
return result;
```

```
CompSym::complement()
1  CompSym result(true,isSet);
2  compAtom thisatom;
3  for compRep.hasMore() do
4      thisatom = compRep.getNext();
5      thisatom.complement();
6      result.intersect(thisatom);
7  this = result;
```

Union : Given two composite formula, A and B, we define A union B as

$$A \vee B = \bigvee_{i=1}^{n_A+n_B} \bigwedge_{j=1}^{t} c_{ij} \tag{3}$$

where for $1 \le i \le n_A$ $c_{ij} = a_{ij}$ and for $n_A + 1 \le i \le n_A + n_B$ $c_{ij} = b_{ij}$. Union of two `CompSym` objects, A and B, is computed by inserting the `compAtom` objects in B to the list of `compAtom` objects in A. The number of disjuncts in the union of two `CompSym` objects, A and B, is $O(n_A + n_B)$.

4 A Polymorphic Verifier

Module `TransSys` in Fig. 2 is responsible for verification. It contains two main functions `check` and `verify` . Check is a recursive function that traverses the syntax tree of CTL formula to compute a symbolic representation for its truth set.

`TransSys` contains following members : `transRelation` (transition relation), `stateSpace` (defined by the domains of the variables in the input specification) and `initialState` (defined by the initial condition of the input specification). These define the transition system for the input specification.

The `verify` function determines whether the given CTL formula is satisfied by the input specification by calling the `check` function. It prints the initial states that violate the formula if the CTL formula is not satisfied by the input specification.

Function `check` is the main part of the module. All computation is done within this function. There are two types of operations : evaluation of logical operators (and, or, not), and evaluation of CTL operators (EX, AX, EF, AF). It assumes that all occurrences of atomic formulas (subformulas with no CTL operators in them) are already converted into `Symbolic` representation. Note that CTL operators that can be expressed in terms of these primitives are first converted into an equivalent representation (e.g. $AG(f) \equiv \neg EF(\neg f)$).

```
Symbolic TransSys::check(Node n) {
  if (n.ofType() == CTLFORMULA)
     switch n.getOperator()
       case AND: s = check(n.left).intersectWith(check(n.right)); break;
       case OR: s = check(n.left).unionWith(check(n.right)); break;
       case NOT: s = check(n.left).complement(); break;
       case NONE: s = check(n.left);
   else if (n.ofType() == CTLOPERATOR)
     s = check (n.left);
     switch n.getOperator()
       case EX:
         s.backwardImage(transRelation);
         break;
       case AX:
         s.complement();
         s.backwardImage(transRelation);
         s.complement();
         break;
       case EF:
         do
           snew = s;
           sold = s;
           snew.backwardImage(transRelation);
           s.unionWith(snew);
         while not sold.isEquals(s)
         break;
```

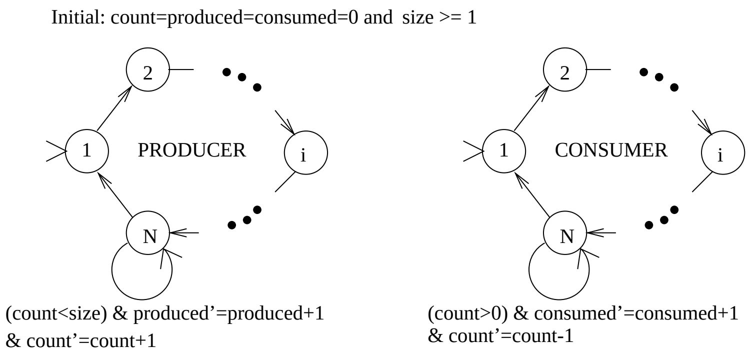

Fig. 4. A simple bounded-buffer producer-consumer example

```
        case AF:
          do
            snew = s;
            sold = s;
            snew.complement();
            snew.backwardImage(transRelation);
            snew.complement();
            s.backwardImage(transRelation);
            s.intersectWith(snew);
            s.unionWith(sold);
          while not sold.isEquals(s)
          break;
    else if (n.ofType() == ATOMIC)
        s = n;
    return s;
}
```

An important feature of function `check` is polymorphism. It is independent of underlying `Symbolic` type. Since each subclass of `Symbolic` implements basic functions (e.g. `intersectWith, backwardImage,` etc.) used, verifier does not need to know which type of representation it is working on. If we introduce a new symbolic type, we do not need to modify the verification procedure. Also, using this feature the verifier can decide which symbolic representation to use at run-time. For example given an input specification with just boolean and enumerated variables, our verifier becomes a BDD-based model checker. Hence, such specifications can be checked efficiently without introducing the cost of manipulating composite representations.

5 A Simple Example

In Fig. 4 we show a simple producer-consumer system. Both *producer* and *consumer* components have N control states. Producer produces an item only when

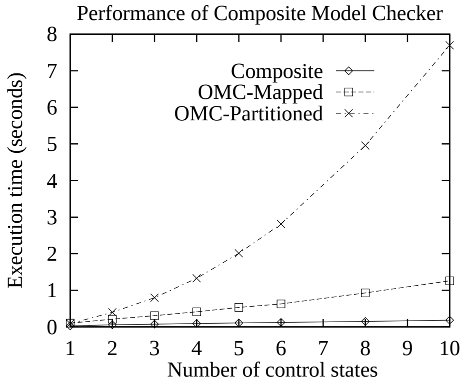

Fig. 5. Performance of composite model checker and Omega Library model checker (using partitioning or mapping approach) on the bounded-buffer producer-consumer example

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 producer and consumer. Self-loop on state N for 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 Figure 4. (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 OMC (Omega Library Model Checker) presented in [BGP97,BGP99]. OMC uses polyhedral representations of arithmetic constraints as a symbolic representation. To represent the control states of the system given in Fig. 4 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 our composite 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 to integers is prevented. Fig. 5 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. 4. Although this is a small example, it demonstrates the inefficiency of using a model checker which is solely based on polyhedral representations.

6 Conclusion and Future Work

The composite symbolic library presented in this paper 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 representations 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.

References

[AB96] J. M. Atlee and M. A. Buckley. A logic-model semantics for SCR software requirements. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 280–292, January 1996.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, X. Nicollin P. H. Ho, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1):3–34, 1995.

[AG93] J. M. Atlee and J. Gannon. State-based model checking of event-driven system requirements. *IEEE Transactions on Software Engineering*, 19(1):24–40, January 1993.

[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.

[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.

[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.

[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 automatic 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.

[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.

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

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

[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.