URBiVA: Uniform Reduction to Bit-Vector Arithmetic

Filip Marić and Predrag Janičić

Faculty of Mathematics, Studentski trg 16, 11000 Belgrade, Serbia
filip@matf.bg.ac.rs   janicic@matf.bg.ac.rs

Abstract. We describe a system URBiVA for specifying and solving a range of problems by uniformly reducing them to bit-vector arithmetic (BVA). A problem description is given in a C-like specification language and this high-level specification is transformed to a BVA formula by symbolic execution. The formula is passed to a BVA solver and, if it is satisfiable, its models give solutions of the problem. The system can be used for efficient modelling (specifying and solving) of a wide class of problems. Several state-of-the-art solvers for BVA are currently used (Boolector, MathSAT, Yices) and additional solvers can be easily included. Hence, the system can be used not only as a specification and solving tool, but also as a platform for evaluation and comparison between BVA solvers.

1 Introduction

In recent years, propositional satisfiability (SAT) and satisfiability modulo theory (SMT) testing have successfully been applied for solving different problems. Huge advances have been made, and state-of-the art SAT and SMT solvers can quickly solve huge problem instances coming from various industrial applications. The progress in this community is strengthen by standardization initiatives like SMT-lib[1] and by annual competitions like SAT-Comp[2] and SMT-Comp.[3] One of the SMT theories that has been extensively used in software and hardware verification lately is the theory of *bit-vector arithmetic (*BVA*)* [4]. Informally, bit-vectors represent fixed-length vectors of bits over which operations are performed as over (finite-precision) integers (either unsigned or two's complement encoded signed). Syntactically, the quantifier-free fragment of the first-order theory of bit-vector arithmetic includes arithmetic operators (`+`, `*`, `-`, `/`, `%`), relational operations (`==`, `!=`, `<`, `>`, `<=`, `>=`), bit-wise operators (`&`, `|`, `^`, `<<`, `>>`), logical operators (`&&`,`||`, `!`), operators for bit-extraction and concatenation, etc. All arithmetic operators are finite-precision and are applied only over bit-vectors of the same width. The semantics of BVA is introduced in a straightforward manner [1]. The satisfiability problem for the quantifier-free fragment of BVA is defined as usual: for a given formula F, check whether there is a variable assignment

[1] http://combination.cs.uiowa.edu/smtlib/
[2] http://www.satcompetition.org/
[3] http://www.smtcomp.org

which makes F true. Although arithmetic over arbitrary precision integers with addition and multiplication is undecidable, BVA is decidable thanks to the finite domain.[4] Additional operators can be defined (in a reasonable manner) without compromising decidability. It can be simply proved that this decidability problem is NP-complete. BVA is suitable for representing and reasoning about programs' properties because it operates on the word-level, in compliance with standard hardware and software operations over integers.

Most (if not all) reported applications of BVA are in the domain of software and hardware verification. We argue that potentials for using BVA solvers are much wider. In this paper, we describe a specification language and a tool UR-BiVA that can be used for solving not just verification problems, but a much wider range of problems (including, for instance, many classes of constraint satisfaction problems). The approach combines features of declarative and imperative programming. It automatically transforms problem specifications to BVA and solves generated formulae by one of underlying BVA solvers. This general reduction approach can be beneficial since hand-crafting BVA formulae that encode specific problems is typically error-prone. As we are aware of, there are still no other tools that uniformly reduce problem specifications to BVA.

2 Problem Specification

The class of problems that are considered are problems of the general form: *find (if it exists) an assignment S which satisfies some given constraints* (variations can require only checking if such an assignment exists, finding all assignments that meet the given conditions, etc). Constraints can be specified by an imperative test that checks whether S (assuming that S is given in advance) is indeed a solution. Therefore, the approach is declarative: only this test is given (instead of a solving procedure). It is often much easier to write such a test, than to write an efficient program that checks satisfiability.

The specification language is C-like and provides all standard arithmetic, bitwise, logical and relational operators, and the ternary conditional operator (`?:`). Two types of variables are supported — numerical (with identifiers starting with `n`) and Boolean (with identifiers starting with `b`). For simplicity, variables are not declared, but introduced dynamically. All operators can be applied to variables of both types (implicit type conversions are performed whenever necessary). User-defined functions are also supported. The `assert` statement specifies a condition (a Boolean expression) that must be satisfied. It triggers the underlying solver to search for an assignment to the unknowns which satisfies the assertion. The `assert_all` statement searches for all such assignments.

Let us illustrate the specification language by considering the problem of finding all Gray codes of given length, i.e., finding all permutations of integers from 0 to $dim - 1$ such that each successive pair of integers differs exactly in one bit in their binary representation. A possible specification of this problem is:

[4] Indeed, for a given formula F to be tested for satisfiability, one can test all possible assignments to variables that appear in F and check if any of them satisfies F.

```
nDim = 8;

bDomain = true;
for (ni = 0; ni < nDim; ni++)
   bDomain &&= 0 <= na[ni] && na[ni] < nDim;

bAllDiff = true;
for (ni = 0; ni < nDim-1; ni++)
   for (nj = ni+1; nj < nDim; nj++)
     bAllDiff &&= na[ni] != na[nj];

bGray = true;
for (ni = 0; ni < nDim - 1; ni++) {
   nDiff = na[ni] ^ na[ni+1];
   bGray &&= !(nDiff & (nDiff - 1)) && (nDiff != 0);
}

assert_all(bDomain && bAllDiff && bGray);
```

The vector `na` is assumed to contain the required permutation (and all such vectors should be found). The code checks if all required conditions are met. The auxiliary variable `bDomain` encodes that all elements of `na` are between 0 and *dim*, `bAllDiff` encodes that all elements of `na` are different, and `bGray` encodes that all successive pairs differ in exactly one bit.

Notice that specifications (implicitly) contain the information on the variables that are *unknown* and have to be assigned so that the given constraints are satisfied. Those are the variables that appear within expressions/statements (not on the left-hand side of the assignment operator) before they were defined (in this example, `na[0]`, ..., `na[7]`). So, the above code is a full and precise specification of the problem, up to the domains of the variables and the semantics of operators (discussed in the next section).

There are certain restrictions of the specification language: conditions in the `if`, `while` and `for` statements and indices for accessing array elements must be ground and not symbolic values. The restriction for `if` is relaxed by the presence of the conditional operator that can take symbolic arguments, while the restriction for arrays and loops cannot be removed (as it would require e.g., undefinite loop unrolling).

3 Problem Solving

Specifications given in the language outlined above are used as a starting point in problem solving. Namely, a problem specification is symbolically executed (for a given fixed bit-width) in order to build a BVA encoding of the problem. The unknowns are represented by BVA variables and results of operations are represented by BVA formulae. Finally, an assertion generates a BVA formula for which a satisfying assignment is to be found. Any satisfying valuation (if it exists) for that formula yields (ground) values for the unknowns that meet the specification, i.e., a solution to the problem.

The semantics of specification language is not equal, but rather parallel to the standard semantics of imperative programming languages. Namely, in the standard semantics, expressions (numerical and Boolean) are always evaluated to ground values and variables must be defined before they are accessed. In the proposed semantics, expressions may be evaluated to ground or symbolic values (BVA formulae) and accessing undefined variables is allowed. In the URBiVA tool, the standard semantics of unsigned BVA is assumed. The domain of Boolean variables is $\{\text{false}, \text{true}\}$ and the domain of numerical variables are finite precision unsigned integers from a domain $[0, 2^l - 1]$, for a given l (so, arithmetic modulo 2^l is assumed).[5] Constant expressions are always evaluated to ground values (for example, after the statement `nA = 3 + 2*5;`, the variable `nA` is assigned the ground value `13`, instead of a BVA formula). Note that even expressions involving symbolic values need not necessarily be evaluated to symbolic values (for example, after the statement `bX = bY && false;`, the variable `bX` can be assigned the ground value *false*, even if the variable `bY` had symbolic value).

Let us illustrate the solving process on the following specification:

```
nB = nA + 3;
nB = 2 * nB;
assert(nA + nB == 12);
```

Since, in the first line of the specification, the variable `nA` was accessed before it was defined, it is associated with a fresh bit-vector variable `A`. In the same line, the formula `A+3` is assigned to `nB`. Similarly, in the second line, the variable `nB` is assigned the symbolic value `2 * (A+3)`. Finally, the `assert` command asserts that `nB + nA == 12` is true, which gives a BVA formula `A + 2*(A+3) == 12` which is tested for satisfiability. It is true if `A` is assigned the value `2`, so a solution to the given problem is `nA == 2` (notice that the variable `nA` was the only unknown in the specification, i.e., only its value is required).

4 Implementation

The system URBiVA[6] is implemented in the programming language C++. The whole system has a flexible architecture and is relatively small. An input specification is parsed into an abstract syntax tree (AST). The interpreter traverses the AST, performs type checking and conversions and executes statements, while keeping a list of unknown variables, and a symbol table containing current variable values. Variable values are represented using a specialized data structure: ground values (BVA constants) are represented by finite length bit-arrays (implemented as byte arrays) and symbolic values (BVA formulae) are represented

[5] The system can be also applied for any finite-precision signed or unsigned, integer or real numbers, as long as the underlying BVA solver provides support for these types.

[6] The source code with example specifications (but without third-party solvers, due to specific licensing) is available online from: `http://argo.matf.bg.ac.rs/software/`.

by term-sharing data structures (DAGs). DAG data structures for representing symbolic values can be either our custom structures, or the ones offered by an underlying solver's API. Using underlying solvers' native data structures helps the integration of the system (and avoids using of external files and textual formats, e.g., SMT-lib). The direct communication via API also facilitates the search for all models: once a model is found, a corresponding blocking clause is constructed and passed to the solver via API and the search for the next model can be (incrementally) started. Currently supported underlying solvers are: our custom solver based on bit-blasting [5] that uses our SAT solver ArgoSAT [6], Boolector[7] [2], Yices[8] and MathSAT[9] [3].

5 Examples and Experimental Results

In this section we give several examples that illustrate the problem modelling and problem solving within the URBiVA system and that we used for a small comparison between the underlying solvers (as yet, larger specifications related to real-world applications have not been considered). We consider one number-theory problem, two combinatorial problems, and one problem from software verification.

Fermat's triples modulo *m*. By the Fermat's last theorem, there are no natural numbers a, b, c such that $a^n + b^n = c^n$ and $n > 2$. However, this does not hold in arithmetic modulo m. The problem of determining the number of solutions of the given equation can be simply stated in our specification language (for a concrete n, say 3) as follows:

```
function nPower(nx, np) {
  nPower=1;
  for(ni = 0; ni < np; ni++)
    nPower *= nx;
}
assert_all(nPower(na,3) + nPower(nb,3) == nPower(nc,3));
```

This specification can solved by the URBiVA system using k-bit representation when arithmetic modulo 2^k is considered.

Gray Codes Problem The Gray codes problem is described and its specification is given in Section 2. The parameter of the problem is dim and it can be solved for different bit-widths (sufficient for storing values from 0 to $dim - 1$).

Magic Square Problem A magic square of order n is a $n \times n$ matrix containing the numbers from 1 to n^2, with each row, column and both diagonals equal the same sum. The problem is to find one (or all, unique up to rotations and reflections) magic square(s) of order n.[10]

[7] http://fmv.jku.at/boolector/

[8] http://yices.csl.sri.com

[9] http://mathsat.itc.it/

[10] For lack of space, we do not give a specification of this problem here, but it is available within the URBiVA distribution.

Software Verification Example — Bit Counting Bit count (or population count) is the problem of counting all set bits of an integer. It can be implemented in a number of ways, two of which are given here for 16-bit integers, specified in our language (almost in verbatim as in the C programming language). The URBiVA tool can be used to show that these two specifications agree on all inputs, i.e., the asserted expression is unsatisfiable.

```
function nBC1(nX) {
  nBC1 = 0;
  for (nI = 0; nI < 16; nI++)
    nBC1 +=  nX & (1 << nI) ? 1 : 0;
}
function nBC2(nX) {
  nBC2 = nX;
  nBC2 = (nBC2 & 0x5555) + (nBC2>>1 & 0x5555);
  nBC2 = (nBC2 & 0x3333) + (nBC2>>2 & 0x3333);
  nBC2 = (nBC2 & 0x0077) + (nBC2>>4 & 0x0077);
  nBC2 = (nBC2 & 0x000F) + (nBC2>>8 & 0x000F);
}
assert(nBC1(nX) != nBC2(nX));
```

Experimental Results. Table 1 shows results of experimental comparison of the four underlying solvers applied on some instances of the four described problems. We solved other instances of these problems and relative performance of the solvers is rather consistent across instance sizes for one problem. We also used different bit-widths for one problem instance and longer bit-widths do not necessarily lead to longer solving times, contrary to what one might expect.

It is interesting to notice that there is no solver superior to others, and that some sorts of problems seem more suited to some solvers (even if the translation mechanism is fixed). This confirms that a system such as URBiVA should take advantage of having several different solvers supported.

Problem	Fermat's triples $n = 3$, bw=6	Gray codes dim=12, bw=4	Magic square $n = 4$, bw=6	Bit Count bw=32
number of solutions	10240	1168	880	0
Boolector	**3.22**	9.37	197.28	**1.20**
MathSAT	98.43	9.72	309.09	*>600.00*
Yices	*144.64*	**2.66**	**76.15**	560.67
bit-blasting	27.18	*12.23*	*461.81*	7.26

Table 1. Results of experimental comparison between four underlying solvers ("bw" denotes bit-width used; all times are given in seconds; best times are given in bold face, worst times are given in italic). All experiments were performed on a PC computer, with Intel Pentium Dual-Core 2.00GHz processor and 2GB RAM.

6 Conclusions, Related Tools, and Further Work

We have described a system that can be used for efficient modelling (specifying and solving) of a wide class of problems by reducing them to BVA and using the power of state-of-the art BVA solvers. The system can also serve as a testing and evaluation platform for BVA solvers. The approach is most suitable for problems for which it is easy to check whether some values satisfy the problems, but it is hard to construct such values. Such problems are, for instance, NP-problems and one-way functions. More generally, the approach can be used for calculating values $\boldsymbol{x}$, such that $f(\boldsymbol{x}) = \boldsymbol{y}$, where f is a function expressible in the described specification language. Still, the approach has two limitations. The first is a finite-precision representation used, and the second is that not all computable functions are expressible since conditional statements and array indices in the specification can involve only expressions that evaluate to ground values.

There is a number of general modelling systems using specific underlying theories and techniques (e.g., CLP(FD) systems, answer set programming (ASP) systems, ILOG OPL, DLV, etc.). All these systems use purely declarative languages (e.g, MiniZinc) and, in contrast to URBiVA, do not have features of imperative programming languages (e.g., destructive assignments). These features, however, make URBiVA directly applicable to wider class of problems (e.g., verification problems). URBiVA is also related to tools for software verification based on symbolic execution (e.g., Java Pathfinder, Pex, SAGE, SmartFuzz, FORTE). Some of these tools use SMT solvers, but they are focused on finding (single) models that lead to bugs (rather than on enumerating all solutions of a given problem). Also, they typically handle only machine data-types (and not arbitrary bit-widths).

URBiVA reduces problems to bit-vector arithmetic. However, the same methodology can be applied, in some cases, for reducing to other SMT problems. We are currently developing a wider system URSA MAJOR, that will use various SMT solvers for various theories, yielding a powerful general problem solving tool. We are planning to consider a wide range of combinatorial and verification problems from various domains and to explore the practical applicability of the approach.

References

1. R. Brinkmann and R. Drechsler. Rtl-datapath verification using integer linear programming. In *Proceedings of the VLSI Design 2002*, IEEE Computer Society, 2002.
2. R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *TACAS*, LNCS 5505, Springer, 2009.
3. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, R. Sebastiani. The MathSAT 4 SMT Solver. In *CAV*. LNCS 5123, Springer, 2008.
4. R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. An abstraction-based decision procedure for bit-vector arithmetic. *STTT*, 11(2), 2009.
5. Predrag Janičić. Uniform Reduction to SAT. manuscript submitted, 2010.
6. Filip Marić. Formalization and Implementation of Modern SAT Solvers. Journal of Automated Reasoning, 43(1), 2009.