URBiVa: Uniform Reduction to Bit-Vector Arithmetic

Filip Marić and Predrag Janičić

Faculty of Mathematics,
University of Belgrade

International Joint Conference on Automated Reasoning 2010

Introduction

- BVA - first order theory of fixed length bitvectors with standard arithmetic (+, −, ∗, /), relational (==,<, >, . . .), bitwise (&, |, . . .), and logical (&&, ||, . . .) operators
- Decidable (due to finite domains of variables)
- Powerful SMT solvers for BVA available
- Used mainly in hardware and software verification
- Potential for wider applications (e.g., constraint satisfaction problems)

Introduction

- BVA - first order theory of fixed length bitvectors with standard arithmetic (+, −, ∗, /), relational (==,<, >, . . .), bitwise (&, |, . . .), and logical (&&, ||, . . .) operators
- Decidable (due to finite domains of variables)
- Powerful SMT solvers for BVA available
- Used mainly in hardware and software verification
- Potential for wider applications (e.g., constraint satisfaction problems)

Motivation

- Problems have to be specified and encoded to BVA so that solvers can be applied
- There are interchangeable formats (e.g., SMT-LIB) but no high-level specification languages
- BVA constraints in applications generated by custom tools
- Generic, high-level modelling system would be welcome

The main ideas of the approach

- Problems considered are of the following general form:
  Find (if they exist) values such that given conditions are met.
- For any candidate values, we can simply test if they make a solution to the problem.
- A test can be written in an imperative language and is a problem specification itself.
- Required values can be automatically found by symbolically executing the test and employing SMT solvers.
- The approach is declarative since no solving procedure needs to be specified

A trivial example

A trivial example

Alice picks a number. She then multipies it by two. Then she inverts the last 4 bits of the obtained result. What is the number that Alice picked, if the obtained result is the same as the initial pick?

Specification

```
nr = na * 2;
nr = nr ^ 0xF;
assert_all(nr == na);
```

- This test is a precise specification of the problem.
- Unknowns (to be determined) are exactly the variables accessed before they were assigned (`na` in this example).

Specification language

- Specification language is C-like
- Numeric and Boolean variables; Arrays.
- All standard operators; `if`, `while`, `for` statements; user defined functions
- `assert` and `assert_all` statements specify the condition that must be met
- Semantics, precisely defined, is different from the standard semantics (e.g., undefined variables can be accessed)
- Restriction: Conditions in loops and array indices must be ground (cannot contain unknowns).

Symbolic execution

- Specifications are symbolically executed
- Result of the interpretation is a quantifier free BVA formula (with unkowns represented as BVA variables)
- SMT solvers are used to find satisfying valuations

Trivial example - cont.

Specification

```
nr = na * 2;
nr = nr ^ 0xF;
assert_all(nr == na);
```

BVA Formula - 8bit

```
na * 2 ^ 0xF == na
```

SMT-LIB format:

```
(= na (bv-xor (bv-mul na 0b00000010) 0b00001111))
```

Solution

The only solution is `na = 0b0000101 = 5`.

Implementation

- Open-source, available at `http://argo.matf.bg.ac.rs`
- flex/bison/C++
- SMT solvers supported: Boolector, Yices, MathSAT
- Communication with SMT solvers using their APIs
- Bit blasting to SAT supported
- All-models feature using blocking clauses

Gray codes example

```
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);
```

```
001
000
010
011
111
110
100
101
```

Bit count example

```
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));
```

Sample experimental results

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

- Different solvers exhibit very different runtime on different instances.
  - Support for multiple solvers is welcome in applications
  - The tool can be used for benchmarking BVA solvers

Related work

- Specification languages for CSP (e.g., MiniZinc, FlatZinc)
  - All declarative (no imperative features, e.g., no destructive assignments)
  - No support for bitwise operators
- Conversions from MiniZinc to SAT [Huang 2008] and SMT [Bofill et al. 2010]
- General modelling systems (e.g., CLP(FD) systems, ASP systems, IBM ILOG OPL)
- Software verification tools based on symbolic execution (e.g., Java Pathfinder, Pex, SAGE, SmartFuzz, FORTE)
  - Not intended for solving combinatorial problems
  - Handle only machine datatypes (no arbitrary bit-width)
  - No all-models functionality

Conclusions

- High level interface to SMT
- New range of applications for BVA (modelling, benchmarking, ...)
- New (imperative-declarative) programming paradigm

Further work

- Tackling some real-world problems
- Overcome some restrictions in the specification language
- URSA Major tool - support for other SMT theories