Uniform Reduction to SAT and SMT

Predrag Janičić  Filip Marić
www.matf.bg.ac.rs/~janicic  www.matf.bg.ac.rs/~filip

Faculty of Mathematics, University of Belgrade, Serbia

COST Action IC0901 Meeting/Third FATPA Workshop
Agenda

- Motivation
- Ongoing Developments
- Specification Language
- Interpretation
- Implementation
- Examples
- Conclusions and Further Work
Motivation

To build a new modelling and solving system (for constraint satisfaction problems, software verification problems, etc.)

High-level specification language should be
- simple but expressible to cover a wide range of problems
- efficiently interfaced with powerful SAT/SMT solvers available

There are interchange formats (e.g., SMT-lib) but no high-level specification languages aiming at SMT

Encoding to SAT/SMT is typically made by special-purpose applications
Ongoing Developments

- Specification language
- Corresponding interpreter
- Link to various SAT/SMT solvers
- Preliminary applications and comparisons
- Still a long way to go
The Basic Idea

- We consider problems of the form: find values that satisfy given conditions
- It is often hard to develop an efficient procedure that finds required values
- It is often easy to specify an imperative test if given values satisfy the constraints
- Such test can be a problem specification itself
- Convert this imperative specification to a SAT/SMT formula and use solvers to search for its models
Simple example

- Alice picked a number and added 3. Then she doubled what she got. If the sum of the two numbers that Alice got is 12, what is the number that she picked?
- A simple test that $A$ is indeed Alice’s number:
  
  $nB = nA + 3$
  $nC = 2 \times nB$
  $assert(nB + nC == 12)$

- This test is a specification of the problem
- **Unknowns** are exactly the variables that were accessed before they were defined
Expressiveness

- The language includes:
  - integer and Boolean data types
  - implicit casting operators
  - arithmetical, logical, relational and bit-wise operators
  - flow-control statements (if, for, while)
- Restriction: conditions in the if, for, while statements must be ground (and not symbolic values)
Interpretation

- Specifications are symbolically executed
- Semantics is different from standard semantics of imperative languages (for instance, undefined variables can be accessed)
- The result of an interpretation is a FOL formula
- This formula is passed to a SAT/SMT solver
- If it is satisfiable, its model will give a solution of the problem
Reduction to SAT/SMT

- Reduction to SAT requires bit-blasting (with a fixed bit-width)
- Reduction to a SMT problem is natural if all relevant operators are supported in the theory (e.g., BVA, LA, UF, ...)
- For bit-vector arithmetic, a fixed bit-width (and hence a finite domain) is used
- Used solvers should be able to give all models of the formula
Simple Example

- Consider the code:
  
  ```c
  nB=nA+3;
nC=2*nB;
assert(nB+nC==12);
  ```

- If $A$ corresponds to the unknown $nA$, then the asserted expression is evaluated to $A + 3 + 2 \times (A + 3) == 12$

- An SMT solver (e.g., for BVA or LA) can confirm that the formula is satisfiable (and is true for $A$ equals 1)
The tool **URSA Major**

- Implemented (in C++) and already fully functional
- It employs a custom subsystem for bitblasting and reduction to SAT
- A SAT solver ArgoSAT and several SMT solvers (MathSAT, Yices, Boolector) for BVA and LA are currently used
URSA MAJOR problem specification
↓ interpreter

Quantifier free FOL formula
↓ bitblasting

Propositional formula
↓ SAT solver  ↓ SMT (BVA, LA, ...) solver

Values of unknowns/Solutions
CSP Example: The Eight Queens Puzzle

```csharp
nDim = 8;
bDomain = true;
bNoCapture = true;
for(ni=0; ni<nDim; ni++) {
    bDomain &&= (n[ni]<nDim);
    for(nj=0; nj<nDim; nj++) {
        if(ni!=nj) {
            bNoCapture &&= (n[ni]!=n[nj]);
            bNoCapture &&= (ni+n[nj]!=nj+n[ni]) && (ni+n[ni] != nj+n[nj]);
        }
    }
}
assert(bDomain && bNoCapture);
```
Verification Example: Bit-counters

function nBC1(nX) {
    nBC1 = 0;
    for (nI = 0; nI < 16; nI++)
        nBC1 += nX & (1 << nI) ? 1 : 0;
}

function nBC2(nX) {
    nBC2 = nX;
    nBC2 = (nc2 & 0x5555) + (nc2>>1 & 0x5555);
    nBC2 = (nc2 & 0x3333) + (nc2>>2 & 0x3333);
    nBC2 = (nc2 & 0x0077) + (nc2>>4 & 0x0077);
    nBC2 = (nc2 & 0x000F) + (nc2>>8 & 0x000F);
}

assert(nBC1(nX)!=nBC2(nX));
Sample Experimental Data

Problem: Magic square, dimension 4
Number of solutions: 880

<table>
<thead>
<tr>
<th>Solver</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yices BVA</td>
<td>76s</td>
</tr>
<tr>
<td>Yices LA</td>
<td>117s</td>
</tr>
<tr>
<td>Boolector BVA</td>
<td>197s</td>
</tr>
<tr>
<td>MathSAT BVA</td>
<td>309s</td>
</tr>
<tr>
<td>bit-blasting</td>
<td>461s</td>
</tr>
</tbody>
</table>
Conclusions

- Applicable to a wide range of problems (e.g., for all NP problems there is a simple witness test)
- Main target: constraint satisfaction problems and software verification problems
- Competitive to other similar systems (e.g., system OPL)
- The approach leads to a new (imperative-declarative) programming paradigm
Further Work

- Support for more SAT/SMT solvers
- Deeper comparison to rival systems
- Real-world applications
- Link to Rich Model Language?