Uniform Reduction to SAT and SMT

Predrag Janičić
www.matf.bg.ac.rs/~janicic
Automated Reasoning GrOup (ARGO)
Faculty of Mathematics
University of Belgrade, Serbia
joint work with Filip Marić

Summer Research Institute (SuRI 2011)
School of Computer and Communication Sciences, EPFL,
Lausanne, Switzerland, June 20, 2011.

Faculty of Mathematics, University of Belgrade

- University of Belgrade
  - Established in early 1800's
  - One of the oldest and largest in the region
  - Around 90000 students and 4000 members of teaching staff
- Faculty of Mathematics
  - Around 1500 students and 80 members of teaching staff
  - Departments for pure mathematics, computer science, astronomy...

Automated Reasoning GrOup (ARGO)

- Area: automated and interactive theorem proving, decision procedures, SAT, SMT, geometry reasoning
- 10 members
- COST Action IC0901 *Rich Model Toolkit* (chair Viktor Kuncak, EPFL)
- SCOPES Joint Research Project *Decision Procedures: from Formalizations to Applications* (with Viktor Kuncak and LARA group, EPFL)
- More at: http://argo.matf.bg.ac.rs/

Motivating puzzle

- Pseudorandom numbers can be generated using linear congruential generators:

$$x_{n+1} \equiv ax_n + c \pmod{m}$$

  where x_0 is the *seed* value ($0 \leq x_0 < m$).
- For example: $x_{n+1} \equiv 1664525x_n + 1013904223 \pmod{2^{32}}$
- Given the seed, it is trivial to compute x_{100}
- Given x_{100}, how to compute the seed?

Problem SAT (SATisfiability)

- Problem of deciding if a given propositional formula in CNF is satisfiable, i.e., if there is assignment to variables such that all clauses are true
- Canonical NP-complete problem
- Example: is $(p \vee q \vee \neg r) \wedge (p \vee \neg q \vee r) \wedge (p \vee \neg q \vee \neg r)$ satisfiable?
- Can be reduced to any NP-complete problem and vice versa
- Very efficient SAT solvers available (typically conflict-driven, clause-learning based)

Problem SMT (Satisfiability Modulo Theory)

- Problem of deciding if a given first-order formula is satisfiable with respect to combinations of background theories
- Examples of theories: linear arithmetic, uninterpreted functions, theories of data structures such as lists, arrays, bit vectors...
- Example: is $x \leq y \wedge y \leq x + c \wedge p(f(x) - f(y)) \wedge \neg p(0)$ satisfiable?
- Very efficient SMT solvers available (typically working in conjunction with SAT solvers)

Reduction to SAT and SMT

- SAT/SMT solvers are widely used, but encoding to SAT/SMT is typically made ad-hoc, by special-purpose tools
- There are interchange formats for SAT/SMT (e.g., SMT-lib) but no high-level specification languages
- No modelling and solving systems based on SMT

Logical analysis of hash functions

- From early 2000's, SAT was used in cryptanalysis
- Hash functions can be explored via SAT
- Rather explore hardness than solve the obtained instances
- Also: useful hard instances can be obtained
- Example problem: for given y, find x such that $hash(x) = y$ (the hash function is *preimage resistant* if this is *hard*)
- How to encode problems as SAT instances?

Fragment of SHA-1 code

```
    for i from 0 to 79
        if 0 ≤ i ≤ 19 then
            f = (b and c) or ((not b) and d)
            k = 0x5A827999
        else if 20 ≤ i ≤ 39
            f = b xor c xor d
            k = 0x6ED9EBA1
        else if 40 ≤ i ≤ 59
            f = (b and c) or (b and d) or (c and d)
            k = 0x8F1BBCDC
        else if 60 ≤ i ≤ 79
            f = b xor c xor d
            k = 0xCA62C1D6

        temp = (a leftrotate 5) + f + e + k + w[i]
        e = d
        d = c
        c = b leftrotate 30
        b = a
        a = temp

    Add this chunk's hash to result so far:
    h0 = h0 + a
    h1 = h1 + b
    h2 = h2 + c
    h3 = h3 + d
    h4 = h4 + e

Produce the final hash value (big-endian):
digest = hash = h0 append h1 append h2 append h3 append h4
```

Encoding cryptanalysis problems

- Obvously, analyzing the code and encoding in SAT by hand – tedious and error prone, instead:
  - use the C-implementation
  - represent variables (that are *unknowns*) by bitvectors (i.e., by vectors of propositional formulae)
  - overload arithmetic operators and run the C++ code (as in symbolic execution)
  - the given constraint evaluates to one propositional formula
  - its model gives the values of the unknowns
- Details: Jovanovic, Janicic: Logical Analysis of Hash Functions, FroCoS 2005.

Toy 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?*
- The computation (if A was given, it just tests if A is indeed the required value)

```
B=A+3;
C=2*B;
assert(B+C==12);
```

- The assertion evaluates to $A + 3 + 2 * (A + 3) == 12$ and further to a SAT instance (if A is represented as a bitvector)

Applicability of the idea

- Given implementation of $f : D \rightarrow D$, and given y, one can compute x such that $f(x) = y$
- Example: the seed problem (the problem can be simply specified and solved, although not efficiently)
- Nice, but is this scope wide? Just for inverting functions?

Applicability of the idea (2)

- Version 1: given $f : D \rightarrow D$ and y one can compute x such that $f(x) = y$
- Version 2: given $f : D \rightarrow \{0,1\}$ one can compute x, if it exists, such that $f(x) = 1$
- Version 3: given $f : D \rightarrow \{0,1\}$ one can check if there is x such that $f(x) = 1$
- Hence, suitable for solving NP-complete problems

Applicability of the idea (3)

- Given f one can check if there is x such that $f(x) = 1$ i.e., check if there are values that satisfy given conditions
- It is often easy to specify an (imperative) test if given values satisfy the conditions (i.e., to express f)
- It is often hard to develop an efficient specialized procedure that finds required values (i.e., to invert f)

Example

- Clique Problem: test whether a given graph contains a clique larger than a given size k
- Hard: check if there is such clique
- Easy: check if a given subgraph is a clique of the size $> k$
- A test if a given subgraph is a clique of the size $> k$ can serve as our specification

Example

- SAT Problem: test whether there is a valuation in which a given formula evaluates to true
- Hard: check if there is such valuation
- Easy: check if in a given valuation the formula evaluates to true
- A test if in a given valuation the formula evaluates to true can serve as our specification

Outline of the system

- Uniform Reduction to SAt
- Stand-alone system, implemented in C++
- C-like specification language
- Unknowns are represented as bit-vectors
- Specifications are symbolically executed
- Constraints give SAT instances
- A model of the SAT instance (if it exists) gives values of the unknowns

Specification Language

- The C-like specification language supports:
  - integer and Boolean data types; arrays
  - implicit casting
  - arithmetical, logical, relational and bit-wise operators
  - flow-control statements (if, for, while) and functions
- Restriction: conditions in `for`, `while` statements and array indices must not contain unknowns

Interpretation

- Specifications are symbolically executed
- The semantics is different from the standard semantics of imperative languages (e.g., undefined variables can be accessed)
- The result of the interpretation is a SAT instance
- If it is satisfiable, its models give solutions of the problem

The seed example

```
nX = nSeed;
for(nI = 0; nI < 100; nI++)
   nX = 1664525 * nX + 1013904223;
assert(nX==123);
```

CSP Example: The Eight Queens Puzzle

```
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));
```

Beyond SAT:URSA Major — Overview of the system

- URSA Major (Uniform Reduction to SAtisfiability Modulo Theory)
- The result of the interpretation is a SAT formula or a FOL formula in a SMT theory
- Basically the same principles as in URSA, but unknowns are not represented as vectors of propositional formula but as theory variables
- Currently several SAT solvers and several SMT solvers used

Overall Architecture

URSA MAJOR problem specification

↓ interpreter

Quantifier free FOL formula

↓ bitblasting

Propositional formula

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

Values of unknowns/Solutions

Additional features w.r.t URSA

- Specifications may involve both interpreted (user-defined) and uninterpreted functions — thanks to the theory of uniterpreted function
- Example:
  `assert(x!=y || f(x)==f(y));`
  where f is not defined

Additional features w.r.t URSA (2)

- Specifications may involve dealing with arrays, thanks to the theory of arrays
- Example:

```
@nA = @nB;
@nB[3]=nX;
assert(@nA == @nB);
```

(`@nA` and `@nB` are arrays)

Comparison with related tools

- Comparable in efficiency to state-of-the-art constraint solvers (e.g., MiniZinc,OPL)
- Can express some constraints that other systems cannot (e.g, constraints over bitvectors, over arrays, involving modular arithmetic)

Comparison with related tools (2)

- One of distinguishing features:
  - The system is declarative (as no solving process has to be specified)
  - The system is imperative (as the specifications are given in the form of imperative tests)
- Can be viewed as a new programming paradigm

Conclusions

- A simple, high-level front-end to SAT/SMT solvers
- Suitable for solving a wide range of problems
- Different real-world applications
- Competitive to other modelling systems
- A novel (imperative-declarative) programming paradigm