Solving XCSP3 constraint problems using tools from formal verification

Martin Mariusz Lester, University of Reading

Automated Program Verification

Does program P satisfy property φ?

We want a **tool** (computer program) to tell us the answer with little or no human intervention.

Testing only catches errors in program execution paths we test.

Verification **should** give us a higher degree of confidence in program correctness.

Verification tools need to employ some kind of **automated reasoning**.

Software Verification Competition (SV-COMP)

Every year, TACAS runs the Software Verification Competition (**SV-COMP**).

ReachSafety track: Given a C program, can it violate an assertion?

Nondeterminism used to model unknown values, such as function arguments.

Programs may include **assumptions** about ranges of variables.

Verification Techniques

Verification tools need to deal with sources of unbounded behaviour:

- Loops, dynamic memory allocation, recursion...

Verification Techniques

Verification tools need to deal with sources of unbounded behaviour:

- Loops, dynamic memory allocation, recursion...

But ultimately, need some kind of automated reasoning or constraint solving:

- Predicate abstraction
- Abstract interpretation
- Translation to automata
- SAT/SMT solvers

How easily can we apply these techniques to constraint problems?

Outline

Automated Program Verification

Verification Tools for Solving Constraint Problems

Implementation: xcsp2c and Exchequer

Is it worth it?

Constraint Programming

Many constraint programming languages available:

- **MiniZinc/FlatZinc** — most popular;
- **Picat** — high-level, more modern;
- **XCSP** — "intermediate" format.

Many approaches available, but usually some combination of propagation, search and heuristics.

Constraint Programming using SAT

PicatSAT won:

- 1st place in XCSP3 Competition 2019;
- 2nd place in MiniZinc Challenge 2021.

SAT is now a leading solution method.

If we didn't have a SAT-based constraint solver, how hard would it be to prototype one?

CBMC (C Bounded Model Checker)

CBMC is a relatively mature software verification tool.

It translates execution of a C program into a giant SAT instance and passes it to a SAT solver.

The instance is satisfiable only if the program can violate an assertion. Possible outcomes:

- **Satisfiable**: The solution to the SAT instance gives a **trace** that leads to the assertion violation.
- **Unsatisfiable**: Program cannot violate an assertion.
- **Timeout**: Don't know.

We can view CBMC as a compiler from C to SAT.

Solving XCSP3 Problems with CBMC

XCSP3 is an XML format for encoding constraint problems.

Idea is to generate a C program that:

- declares all problem variables;
- set variables nondeterministically;
- `assumes` all constraints and `asserts` false.

Pass to CBMC and get back a solution.

Example: XCSP3 instance and C encoding

```
<instance format="XCSP3" type="CSP">
  <variables>
    <var id="x"> 1..10 </var>
    <var id="y"> 1..100 </var>
  </variables>
  <constraints>
    <intension>
      eq(y,mul(x,x))
    </intension>
  </constraints>
</instance>
```

```
int main() {
    int32_t x;
    __CPROVER_assume(((x >= 1) && (x <= 10)));
    __CPROVER_printf("XCSP2C SOLUTION: x = %d", x);
    int32_t y;
    __CPROVER_assume(((y >= 1) && (y <= 100)));
    __CPROVER_printf("XCSP2C SOLUTION: y = %d", y);
    __CPROVER_assume((y == (x * x)));
    assert(0); }
```

Implementation: xcsp2c and Exchequer

Prototype of transformation implemented in `xcsp2c`.

Solver **Exchequer** submitted to mini track of XCSP3 competition.

Is it worth it?

Performance

Hypothesis: Exchequer/xcsp2c is acceptable, but worse than PicatSAT.

Obviously, the translation adds overhead not present in a direct translation.

We should not expect it to be better.

How much worse? Wait for XCSP3 Competition results this week.

Insight gained

Found a bug in CBMC:

```
__CPROVER_assume(
        ((x1 == 0) && (x2 == 1))
    ||  ((x1 == 0) && (x2 == 2))
    ||  ((x1 == 1) && (x2 == 5))
    ...
);
```

Workaround:

```
{
    int32_t flag = 0;
    flag = flag || (x1 == 0) && (x2 == 1);
    flag = flag || (x1 == 0) && (x2 == 2);
    flag = flag || (x1 == 1) && (x2 == 5);
    __CPROVER_assume(flag);
}
```

Insight gained

For many benchmark problems with large `extension` constraints, CBMC spent much more time translating C into SAT than calling SAT solver.

The C code is essentially a "straight line" program. It should be simple to translate.

Is this because of short-circuiting?

```
flag = flag || (x1 == 0) && (x2 == 1);
```

No. Same problem with bitwise operations:

```
flag = flag | (x1 == 0) & (x2 == 1);
```

(Compare results of Verma and Yap.)

Insight gained

Problem: Normally, CBMC needs to compute bounds on loops, so it knows how much to unroll them.

To do this, it uses abstract interpretation on variables.

With many expressions with many variables, this becomes slow.

Prototyping new techniques

Primary motivation was to develop an easy route to prototyping already-implemented techniques from software verification.

The same C translation can be used with multiple solvers, but:

One motivation for XCSP3 was to provide a format that retains high-level structure of a constraint problem.

Translation for CBMC aimed to keep the translated code as simple as possible to keep the translation to SAT fast.

No use of arrays or functions.

Maintaining high-level structure

Many verification tools use predicate abstraction (or similar) to try to infer loop invariants or function preconditions/postconditions.

If we have a group of XCSP3 constraints, applied to different variables, may want to encode this as several calls to a C function.

Similarly, may wish to encode XCSP3 array as C array.

But this will make the problem harder for other tools!

Also, XCSP3 encoding has often already removed some useful structure by "unrolling" constraint groups!

Conclusion

I have implemented a prototype translation from XCSP3 to C (xcsp2c) and associated constraint solver (Exchequer).

Benchmarking is ongoing.

Ideally, we want to use other verification tools and exploit their high-level reasoning.

But this may be difficult in practice.

Thanks for Listening

Any Questions?