Topic 10: Modelling for SAT and SMT
(Version of 22nd February 2018)

Jean-Noël Monette

Optimisation Group
Department of Information Technology
Uppsala University
Sweden

Course 1DL441:
Combinatorial Optimisation and Constraint Programming,
whose part 1 is Course 1DL451:
Modelling for Combinatorial Optimisation

UPPSALA UNIVERSITET

Outline

1. SAT and SMT

2. Encoding into SAT

3. Modelling for SAT and SMT in MiniZinc

4. Case Study
   Graph Colouring

UPPSALA UNIVERSITET

Outline

Revisit the slides on Boolean satisfiability (SAT) and SAT modulo theories (SMT) of Topic 7: Solving Technologies.

SAT using MiniZinc:

The currently only SAT backend, `mzn-g12sat`, allows only Boolean decision variables:

- One must manually transform a model with integer variables (see the next slides), just like when directly using a SAT solver. This will probably be automated in a future release of the MiniZinc toolchain.
- One can already flatten a model with set variables using the option `-Gnosets`.

SMT using MiniZinc:

- The backend `fzn2smt` transforms a FlatZinc model into the SMTlib language, which is understood by most SMT solvers, such as CVC4, Yices, and Z3.
- All predicates are decomposed by the flattening.

UPPSALA UNIVERSITET

Outline

UPPSALA UNIVERSITET

Encoding into SAT

Challenges:

- How to encode an integer variable into a collection of Boolean variables?
- How to encode a constraint on integer variables into a collection of constraints on Boolean variables?
- How to transform a constraint on Boolean variables into clausal form? Most solvers do this for you.

Considerations:

- We want few variables.
- We want few clauses.

As usual, there are many possibilities and it is not always clear what is the best choice.

Encoding an Integer Variable

Well-known encodings, described on the next three slides:

- Direct (or sparse) encoding:
  a Boolean variable for each equality with a value.
- Order encoding:
  a Boolean variable for each inequality with a value.
- Direct + order encoding:
  channel between the direct and order encodings.
- Bit (or binary) encoding:
  a Boolean variable for each bit in the base-2 representation of the domain values [not covered here].

Direct Encoding of an Integer Variable

Consider an integer variable x with domain $1..n$:

- Create a Boolean variable $b_{[x=k]}$ for all k in $1..n$.
- The variable $b_{[x=k]}$ is **true** if and only if $x = k$ holds.
- Consistency constraints:
  - At least one value: $\bigvee_{k \in 1..n} b_{[x=k]}$
  - At most one value: $\bigwedge_{j,k \in 1..n,\ j<k} \neg \left(b_{[x=j]} \wedge b_{[x=k]} \right)$
- There are n variables, $n \cdot (n-1)/2$ binary clauses, and one n-ary clause.
- The constraint $x \neq k$ is encoded as $\neg b_{[x=k]}$.
- The constraint $x < k$ is encoded as $\bigwedge_{j \in k..n} \neg b_{[x=j]}$.

Order Encoding of an Integer Variable

Consider an integer variable x with domain $1..n$:

- Create a Boolean variable $b_{[x \geq k]}$ for all k in $1..(n+1)$.
- The variable $b_{[x \geq k]}$ is **true** if and only if $x \geq k$ holds.
- Consistency constraints:
  - Order: $\bigwedge_{k \in 1..n} (b_{[x \geq k]} \vee \neg b_{[x \geq k+1]})$
  - Bounds: $b_{[x \geq 1]} \wedge \neg b_{[x \geq n+1]}$
- There are $n+1$ variables and n binary clauses.
- The constraint $x = k$ is encoded as $b_{[x \geq k]} \wedge \neg b_{[x \geq k+1]}$.
- The constraint $x \neq k$ is encoded as $\neg b_{[x \geq k]} \vee b_{[x \geq k+1]}$.
- The constraint $x < k$ is encoded as $\neg b_{[x \geq k]}$.

Direct + Order Encoding of Integer Variable

- Channelling constraints:

$$\bigwedge_{k \in 1..n} \left(b_{[x=k]} \Leftrightarrow \left(b_{[x \geq k]} \wedge \neg b_{[x \geq k+1]} \right) \right)$$

- Reminder: $\alpha \Leftrightarrow \beta$ is equivalent to $(\neg\alpha \vee \beta) \wedge (\alpha \vee \neg\beta)$.
- Channelling constraints in clausal form:

$$\bigwedge_{k \in 1..n} \left((\neg b_{[x=k]} \vee b_{[x \geq k]}) \wedge (\neg b_{[x=k]} \vee \neg b_{[x \geq k+1]}) \right.$$
$$\left. \wedge \, (b_{[x=k]} \vee \neg b_{[x \geq k]} \vee b_{[x \geq k+1]}) \right)$$

- There are $2 \cdot n$ new binary and n new ternary clauses.

Encoding a Constraint of Arity > 1

- A constraint encoding is a constraint decomposition in clausal form.
- Many possibilities exist for each constraint predicate.
- There is a lot of research on how to encode constraints.
- There are two general approaches:
  - encode solutions;
  - encode non-solutions.
- Constraint encodings depend on the variable encoding.

Encoding Simple Constraints of Arity > 1

Constraint $x = y$, both variables with domain $1..n$:

- Direct encoding: $\bigwedge_{k \in 1..n} (b_{[x=k]} \Leftrightarrow b_{[y=k]})$
- Order encoding: $\bigwedge_{k \in 1..n} (b_{[x \geq k]} \Leftrightarrow b_{[y \geq k]})$

Constraint $x \neq y$, both variables with domain $1..n$:

- Direct encoding: $\bigwedge_{k \in 1..n} (\neg b_{[x=k]} \vee \neg b_{[y=k]})$
- Order encoding:

$$\bigwedge_{k \in 1..n} (\neg b_{[x \geq k]} \vee b_{[x \geq k+1]} \vee \neg b_{[y \geq k]} \vee b_{[y \geq k+1]})$$

Constraint $x \leq y$, both variables with domain $1..n$:

- Direct encoding: $\bigwedge_{k \in 1..n} \left(\neg b_{[x=k]} \vee \bigvee_{j \in k..n} b_{[y=j]} \right)$
- Order encoding: $\bigwedge_{k \in 1..n} \left(\neg b_{[x \geq k]} \vee b_{[y \geq k]} \right)$

Constraint $x + c = y$, with $x \in 1..n$ and $y \in (1+c)..(n+c)$:

- Direct encoding: $\bigwedge_{k \in 1..n} \left(b_{[x=k]} \Leftrightarrow b_{[y=k+c]} \right)$
- Order encoding: $\bigwedge_{k \in 1..n} \left(b_{[x \geq k]} \Leftrightarrow b_{[y \geq k+c]} \right)$

Encoding `alldifferent`

How to encode `alldifferent`$([X_1, \ldots, X_m])$, where all the variables have domain $1..n$, with usually $m \leq n$?

- Approach 1: Decompose `alldifferent` into binary disequalities, and encode each disequality separately.
- Approach 2: Each value is taken by at most one var:
  - Use a decomposition into binary clauses, like in the direct encoding: this actually boils down to Approach 1.
  - Use a ladder encoding: see the next slide.

Ladder Encoding for `alldifferent`

- Add a Boolean variable A_{ik} for each $i \in 0..m$ & $k \in 1..n$.
- Variable A_{ik} is **true** iff one of $X_1, \ldots, X_i$ has value k.
- Constraints:
  - Consistency:

$$\bigwedge_{k \in 1..n} \bigwedge_{i \in 1..m} \left(\neg A_{(i-1)k} \vee A_{ik} \right)$$

  - Channelling:

$$\bigwedge_{k \in 1..n} \bigwedge_{i \in 1..m} \left(B_{[X_i = k]} \Leftrightarrow \left(\neg A_{(i-1)k} \wedge A_{ik} \right) \right)$$

Comparison of `alldifferent` Encodings

Decomposition encoding:

- The decomposition has $\frac{m \cdot (m-1)}{2}$ binary disequalities.
- Each disequality is encoded by n binary clauses.
- Total: $\frac{n \cdot m \cdot (m-1)}{2}$ binary clauses.

Ladder encoding:

- First part: $n \cdot m$ binary clauses.
- Second part: $2 \cdot n \cdot m$ binary and $n \cdot m$ ternary clauses.
- Total: $4 \cdot n \cdot m$ clauses with 2 or 3 literals.

UPPSALA UNIVERSITET

Outline

Some Guidelines

Use higher-level variable types (sets, . . .),
even if not supported by the SAT or SMT backend.

- They enable the use of carefully designed encodings.
- It is easier for the modeller to reason about them.

Use global constraint predicates,
even if not supported by the SAT or SMT backend.

- They enable the use of carefully designed encodings.
- It is easier for the modeller to reason about them.

Add implied constraints:

- They may reduce the search space a lot
- This might be redundant with the clause-learning mechanisms of modern SAT and SMT solvers.

Add symmetry-breaking constraints:

- They may reduce the search space a lot.

Limit the number of SMT theories used:

- Using several theories may decrease the inference power of the solver.

UPPSALA UNIVERSITET

Outline

1. SAT and SMT

2. Encoding into SAT

3. Modelling for SAT and SMT in MiniZinc

4. Case Study

Graph Colouring

UPPSALA UNIVERSITET

Outline

1. SAT and SMT

2. Encoding into SAT

3. Modelling for SAT and SMT in MiniZinc

4. Case Study

Graph Colouring

Given a graph `(V,E)`, colour each vertex so that adjacent vertices have different colours, using at most `n` colours.

Model 2, MiniZinc/CP/LCG-style

Variable `C[v] in 1..n` is `k` iff vertex `v` has colour `k`.

1. One colour per vertex: enforced by choice of variables.
2. Different colours on edge ends:

```
forall((u,v) in E)(C[u] != C[v])
```

Give SAT and SMT flattenings of this model!