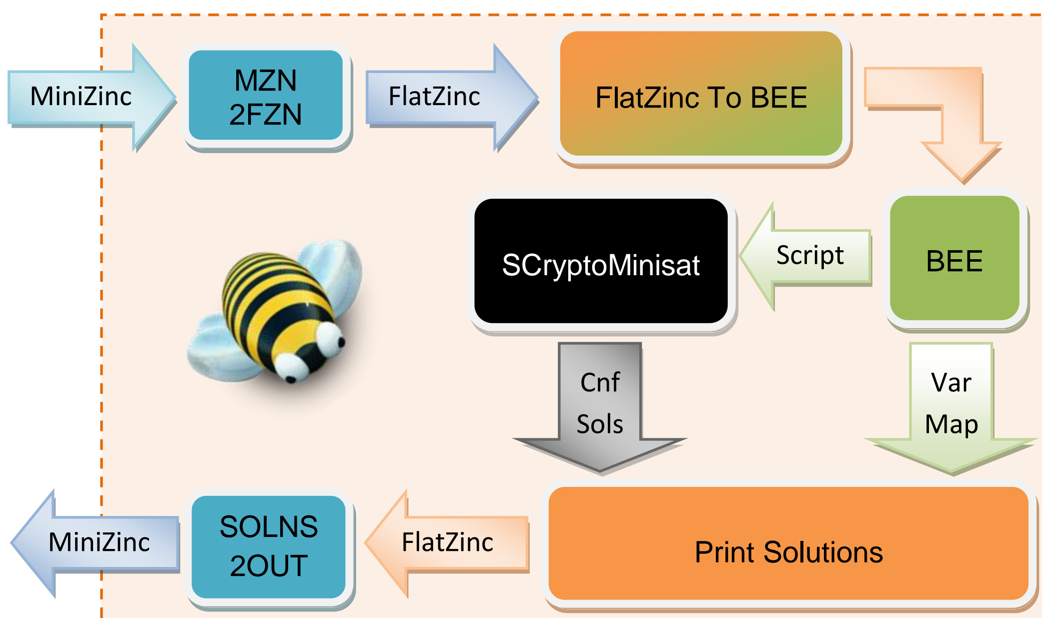
BumbleBee

SAT based MiniZinc/FlatZinc Solver

Amit Metodi, Yoav Fakete, and Michael Codish

BumbleBee is a MiniZinc/FlatZinc solver based on an encoding to SAT which is composed from the following chain of tools:



These components are written in different languages (such as Prolog, Python, and C++) and implemented as individual black boxes. The components are "glued" together using. The black box components include: MZN2FZN & SOLNS2OUT (Blue in the figure) , *BEE* (Green in the figure), and SCryptoMinisat (Black in the figure) .

MZN2FZN & SOLNS2OUT

Responsible to convert between MiniZinc and FlatZinc.

These components are part of the G12 MiniZinc Distribution which can be found here: http://www.g12.csse.unimelb.edu.au/minizinc/download.html

Note: When applying MZN2FZN we maintain the following few global constraints:

- all_different_int(array[int] of var int: x)
- minimum_int(var int: m, array[int] of var int: x)
- maximum_int(var int: m, array[int] of var int: x)

BEE – *B*en-Gurion-university *E*qui-propagation *E*ncoder

BEE is a constraint simplifier and encoder. It receives a list of constraints and returns their CNF. Currently, it represents each integer variable as a unary number (order encoding). Currently, simplification of constraints is performed using ad-hoc rules for Boolean equi-propogation on each constraint.

More details about Boolean Equi-Propogation can be found in the following paper: "Boolean Equi-propagation for Optimized SAT Encoding - Amit Metodi, Michael Codish, Vitaly Lagoon, Peter J. Stuckey" (http://arxiv.org/abs/1104.4617)

BEE is implemented in Prolog and therefore encoding instances involving a large number of constraints increases encoding time. Numbers are represented in unary which can dramatically effect the system and sometimes cause it to fail because of large memory usage.

***SCryptoMinisat* – Script SAT Solver**

SCryptoMinisat (pronounce this as "scripto – miniSAT") is an our extension of the CryptoMinisat 2.5.1 sat solver which enables the user to write scripts involving successive calls to the solver. It addresses two main needs:

a) Minimizing (or Maximizing): Given a unary number and a CNF instance, repeatedly call the SAT solver to find a satisfying solution which minimizes (or maximizes) the value of the number. C
b) All solutions: asking a SAT solver to find all solutions for a specific set of literals of interest, usually means that after each solution we add a clause to negate that those literals of the solution when seeking the next.

SCryptoMinisat adds the following features:

- Declare relevant literals
  - Will output only relevant literals assignments.
  - When searching for all solutions, each solution will have different assignment for relevant literals.
- Minimize function

Receives a lists of literal representing a unary number and it will solve the instance to minimize that number.

Python Glue (includes FlatZinc To BEE & Print Solutions)

The Python glue is responsible to execute the different tools and coordinate between them.

FlatZinc To BEE

This tool has two parts:

The Python part which performs a syntactic transformation from FlatZinc code, to an equivalent Prolog code.

The Prolog part which using the code generated in the Python part performs type checking, finding ranges for unbounded variables (if possible) and decides the underlining representation of the FlatZinc variables.

In the end of this step a list of *BEE* constraints is generated which are equivalent to FlatZinc constraints and passed to *BEE*.

Print Solutions

This Python part is responsible to print FlatZinc solutions from the BEE variable map and SCryptoMinisat satisfied assignments. (In case the original problem was a MiniZinc problem the SOLNS2OUT is used to convert FlatZinc solutions to MiniZinc solutions).

Unbounded variables handling

The Python glue is also responsible to handle instances with unbounded variables. In case during FlatZinc to BEE it found that at least one variable is unbounded we apply the iterative deepening algorithm on the range of the unbounded variables. It is done in order to make the solver complete but it won't necessarily terminate. For example for when minimizing an unconstrained variable the solver will probably never stop.