A General Local Search Solver for FlatZinc

Sara Ceschia[1], Luca Di Gaspero[1], Andrea Schaerf[1], Tommaso Urli[2]

[1] DIEGM, University of Udine
Via delle Scienze 206, 33100 Udine, Italy
{sara.ceschia,luca.digaspero,andrea.schaerf}@uniud.it

[2] Optimization Research Group, NICTA and Australian National University (ANU)
7 London Circuit, Canberra ACT 2601, Australia
tommaso.urli@nicta.com.au

1 Introduction

MiniZinc [5, 7] is a high-level declarative modeling language that has become quite popular in the last few years. One of the main features of MiniZinc is the underlying middle-level constraint language FlatZinc, into which a MiniZinc model, along with a given instance, is translated.

The generality of MiniZinc, and the low-level nature of FlatZinc, which allows it to be implemented efficiently by many solvers, have generated a lot of interest in the CP community around the MiniZinc/FlatZinc tool chain. The flexibility and expressivity of the MiniZinc language is proved by the large variety of modeling constructs that are supported: it includes arrays and sets, along with the the basic types (`int`, `float`, `bool`) for parameters (i.e., constants) and decision variables, conditional and loop statements, arithmetic expressions (plus, times, linear equations, inequalities, . . .), boolean expressions (conjunction, disjunction, implication, negation), global constraints (e.g., `alldifferent`) and set constraints, overloading of built-in and user-defined operations.

Quite a few FlatZinc solvers are publicly available in the research community, but, up to our knowledge, none of them uses meta-heuristics. Among these, GECODE [10] and JACOP [6] are pure constraint programming (CP) solvers, SICStus Prolog [11] has a Prolog engine, MiniSAT(ID) [9] implement propositional logic (SAT) solving techniques, SCIP [1] and Opturion CPX [8] are hybrid solvers: Opturion CPX combines CP and SAT, SCIP is mainly a Mixed Integer Programming (MIP) solver, but it incorporates also a branch-&-cut-&-price and CP framework with SAT-solving features. Currently, the only attempt to design a local search back-end for MiniZinc can be ascribed to Biordal [2] who translate the FlatZinc into a Constraint-Based Local Search model.

In this work, we describe an on-going project consisting in the implementation of a FlatZinc solver based on local search. The solver makes use of the framework EASYLOCAL++ [4], which provides an abstract implementation of local search techniques. The system is currently at its early stage of development. As such, it implements only integer decision variables and uses a simple "change value to one variable" neighborhood relation. We test the system on the Curriculum-based Course Timetabling problem and compare its results with available FlatZinc solvers. The outcome is quite encouraging as the system, already in the present form, is at the same level of the best available solvers.

2 Local Search Solver for FlatZinc

The overall translation process from FlatZinc to an EASYLOCAL++ solver is illustrated in Figure 1. The local search solver for FlatZinc is based on the new modeling capabilities available in the 3.0 version[1] of EASYLOCAL++. The modeling layer allows to define a high-level description of an optimization problem in terms of decision variables and algebraic expressions which are then compiled in an intermediate form that allow an efficient evaluation of local search moves, based only on the modified variables.

A FlatZinc file consists of four parts: *i) parameter declarations*, *ii) variable declarations*, *iii) constraints*, and *iv) solve goal*. Besides the parameters, whose values are directly expanded while parsing

[1] Available at https://bitbucket.org/satt/easylocal-3

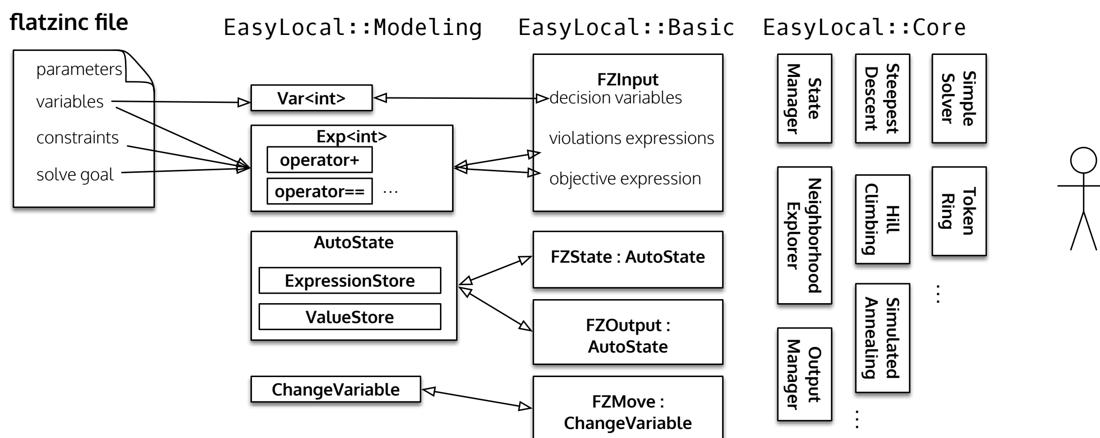

Figure 1: Translation of a FlatZinc file to an EasyLocal executable solver.

the FlatZinc file, the other components are translated into `EasyLocal::Modeling` constructs. In particular, decision variables become variable objects of EASYLOCAL++'s modeling layer whereas intermediate (i.e., introduced) variables and constraints are translated into algebraic expressions. In the case of constraints, the expressions represent a *violation measure* of the corresponding constraint.

For example, the FlatZinc constraint $\texttt{int_plus}(x, y, 5)$, which states that the sum of x and y should be equal to 5, is translated into the corresponding violation expression $x + y \neq 5$ whose truth value is computed during the search. The solve goal might include an optimization variable that is bound to an objective expression.

Local search upon the `EasyLocal::Modeling` representation is based on a search space that associates a value to each decision variable and on a neighborhood definition that changes the value of one decision variable at the time. These components are the basis for building an EASYLOCAL++ solver exploiting the different meta-heuristic components and high-level composition strategies made available by the framework.

3 Case Study

We have implemented two MiniZinc models for the Curriculum-based Course Timetabling problem. For the sake of brevity we refer to [3] for the definition of the problem.

The first model considers only the hard constraints and uses two arrays of variables, which assign each lecture to a period and to a room, respectively. The second model includes also all the soft constraints that compose the objective function, and uses several auxiliary variables for the definition of the various components to minimize.

We experimented our solver both on the 21 `comp` instances of the ITC 2007 challenge and on a group of new artificial instances created using the generator by Leo Lopes [12], which is parameterized upon the total number of lectures and the percentage of room occupation. In detail, we generated 5 instances for each value of the number of lectures in the range $\{i \cdot 10 | i = 1, \ldots, 13\}$, fixing the percentage of occupation to 70%. We compare our solver with the FlatZinc solvers available in the distribution of MiniZinc (v.1.6): GECODE (v.4.3.3), JACOP (v.4.2.0) and Opturion CPX (v.1.0.2). The results for the first model are reported in Table 1 in terms of percentage of feasible solutions found within the granted computational time (5 minutes on our machine). For our stochastic local search solver, we perform 30 runs for each instance. The average running times of our solver were 6 seconds for the `L*` family and 24 seconds for the `comp` family. The table shows that our solver performs at the same level of the best available ones, which however are exact.

For the complete model it is necessary to identify functionally dependent variables, in order to reduce the search space. Experiments on the different strategies for this specific task are under evaluation.

Table 1: Results of different FlatZinc solvers: percentage of feasible solutions found within the timeout.

Instance	minizinc	mzn-g12cpx	mnz-g12fd	mnz-g12lazy	Gecode	JaCoP	Opturion CPX	Local Search
L10-70-0/4	100%	100%	100%	100%	100%	100%	100%	100%
L20-70-0/4	100%	100%	100%	100%	100%	100%	100%	100%
L30-70-0/4	0%	100%	0%	100%	0%	0%	100%	100%
L40-70-0/4	0%	40%	0%	100%	0%	0%	100%	100%
L50-70-0/4	0%	80%	0%	100%	0%	0%	100%	100%
L60-70-0/4	0%	80%	0%	100%	0%	0%	100%	100%
L70-70-0/4	0%	40%	0%	100%	0%	0%	100%	100%
L80-70-0/4	0%	60%	0%	100%	0%	0%	100%	100%
L90-70-0/4	0%	20%	0%	100%	0%	0%	100%	100%
L100-70-0/4	0%	0%	0%	100%	0%	0%	100%	100%
L110-70-0/4	0%	0%	0%	100%	0%	0%	100%	99%
L120-70-0/4	0%	20%	0%	100%	0%	0%	80%	100%
L130-70-0/4	0%	0%	0%	100%	0%	0%	100%	100%
comp1/21	4%	0%	4%	61%	26%	0%	13%	19%

References

[1] Tobias Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1:1–41, 2009.

[2] Gustav Biordal. The First Constraint-Based Local Search Backend for MiniZinc. Master's thesis, Uppsala University, 2014.

[3] Alex Bonutti, Fabio De Cesco, Luca Di Gaspero, and Andrea Schaerf. Benchmarking curriculum-based course timetabling: formulations, data formats, instances, validation, visualization, and results. *Annals of Operations Research*, 194(1):59–70, 2012.

[4] Luca Di Gaspero and Andrea Schaerf. EASYLOCAL++: An object-oriented framework for flexible design of local search algorithms. *Software—Practice and Experience*, 33(8):733–765, 2003.

[5] NICTA Optimization Research Group. Minizinc website. URL: http://www.minizinc.org/. Viewed: 2nd February 2015.

[6] Krzysztof Kuchcinski and Radoslaw Szymanek. JaCop website. URL: http://jacop.osolpro.com/. Viewed: 2nd February 2015.

[7] Nicholas Nethercote, Peter J Stuckey, Ralph Becket, Sebastian Brand, Gregory J Duck, and Guido Tack. MiniZinc: Towards A Standard CP Modelling Language. In *Principles and Practice of Constraint Programming CP 2007*, pages 529–543, 2007.

[8] Opturion. Opturion CPX website. URL: http://www.opturion.com/cpx.html. Viewed: 2nd February 2015.

[9] Knowledge Representation and Reasoning group (Katholieke Universiteit Leuven. MinisatID website. URL: http://dtai.cs.kuleuven.be/krr/software/minisatid. Viewed: 2nd February 2015.

[10] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode/Flatzinc website. URL: http://www.gecode.org/flatzinc.html. Viewed: 2nd February 2015.

[11] Computer Systems Laboratory (SICS). SICStus Prolog website. URL: https://sicstus.sics.se/. Viewed: 2nd February 2015.

[12] K. A. Smith-Miles and L. Lopes. Generalising algorithm performance in instance space: A timetabling case study. In C. A. Coello Coello, editor, *Learning and Intelligent Optimization (LION 2011)*, number 6683 in Lecture Notes in Computer Science, pages 524–538. Springer, 2011.