Universitat de Girona
Escola Politècnica Superior

Treball final de grau

Estudi: Grau en Enginyeria Informàtica

Títol:

GOS
A new declarative tool for modelling and solving *CSPs to SAT*

Document: Summary

Alumne: Roger Generoso Masós

Tutor: Jordi Coll i Mateu Villaret
Departament: Informàtica, Matemàtica Aplicada i Estadística
Àrea: Llenguatges i Sistemes Informàtics

Convocatòria (mes/any)

Juny 2020

1 Introduction

Constraint Satisfaction Problems (CSP)s consists on finding values for a set of variables subject to a set of constraints. Examples of CSPs are well-known puzzles such as *Sudoku* or problems appearing in industry such as *Scheduling* or *Timetabling*. This kind of problems can be easily modelled with declarative programming languages.

Declarative programming languages attempt to describe what the program must accomplish in terms of the problem domain, rather than describe how to accomplish it as a sequence of the programming language primitives. This is in contrast with imperative programming, which implements algorithms in explicit steps.

A subset of declarative languages are *modelling* languages. This project will be focused on this subset and the main purpose will be get a new declarative programming language for modelling any CSP to Boolean Satisfiability (SAT).

One of the most successful methodologies for solving CSP relies on the conversion into SAT problems, i.e., to build a Boolean formula that has a solution if and only if the original CSP has also a solution. The advantage is the wide availability of free and efficient SAT-solvers.

Currently there exist some declarative languages to model CSPs, such as *MiniZinc* (Stuckey et al., 2018), *ESSENCE*' (Frisch et al., 2008), *Picat* (Zhou & Kjellerstrand, 2016), *WSimply* (Ansótegui et al., 2013) and many others, which have their own compiling and solving systems. Some of these systems support automatic reformulation to SAT. However, one of the current research lines of the Logic and Programming (L∧P) research group of the *University of Girona* is to find efficient SAT encodings of particular constraints. For this reason, the L∧P group is interested in having its own SAT declarative modelling language and compiler that can be directly integrated with their SAT encoding systems.

The name chosen for the tool is *GOS* (**G**irona **O**ptimization **S**ystem) and the language that *GOS* will use is *BUP*.

1.1 Purpose

The main purpose of the project is to obtain a new declarative programming language for modelling any CSP to SAT. To achieve this, it will be necessary to think and define a new programming language allowing the user define CSPs as SAT in a declarative way, with a higher level of expressiveness, and implement a compiler that integrates a SAT-solver to obtain the solution of the modelled problems. The goal is to provide an easy gateway for modelling with SAT, that has extremely efficient solvers. Figure 1 is a high level schema of the project.

1.2 Objectives

To reach the main purpose, many objectives must be satisfied:

- Design a programming language to improve the expressiveness when encoding any CSP to SAT.

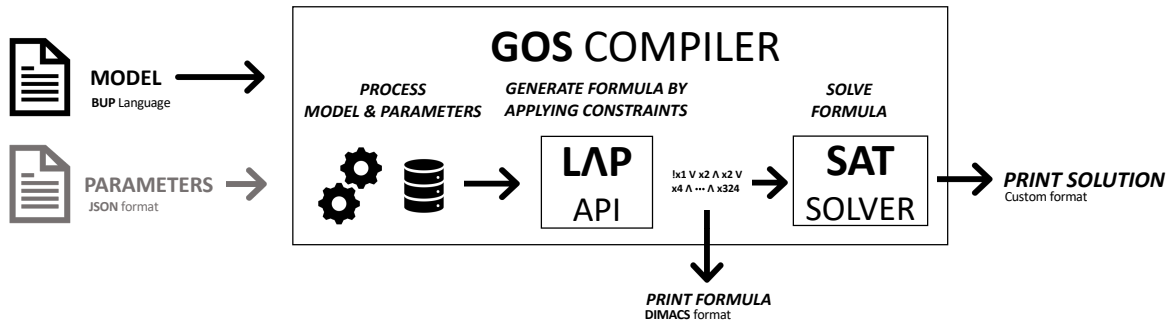

Figure 1: High level *GOS* tool schema

- Implement a compiler for this language to be able to parse any input and give it semantic meaning.
- Integrate the compiler with the API implemented by L∧P group in order to use the solvers to solve the modelled problems.

My personal objectives developing this project are get a deepen knowledge in how compilers work, SAT encoding and declarative and formal languages.

2 Working framework

This project covers two main disciplines: development of compilers and SAT modelling.

2.1 Compiler development

The *GOS* compiler has been implemented using Another Tool for Language Recognition (ANTLR) 4, a parser generator for reading, processing, executing, or translating structured text or binary files. The main tasks related to develop the compiler have been: defining lexical rules to recognize the different tokens of the *BUP* language; define syntactic rules to build the parsing tree of a *BUP* program; implement the semantic correctness checks of a *BUP* program; implement the translation from the constraints specified in *BUP* language to a *SAT* formula.

2.2 SAT modelling

A SAT problem contains a formula built on a set of boolean variables, which can take only value *true* (or 1) and *false* (or 0). A solution to SAT problem is an assignment of values *true/false* to the logical variables, such that all clauses are satisfied. It is well known that SAT can be a very efficient approach to model and solve CSPs. However, its main handicap is the lack of expressiveness, since only propositional formulas are supported. For this reason, the *BUP* language has been designed to support more expressive structures. A good understanding of the SAT problem has been required in this project in order to define the *BUP* language and translate *BUP* models to SAT formulas.

3 Results

3.1 *BUP* programming language

BUP is a new declarative programming language for modelling *CSP* and solve them using SAT. As far as we know, there are no precedents of declarative modelling languages allowing to define tuples. *BUP* permits create complex data structures (called *entities*) that allow you to group params and variables in a common framework.

To sum up, *BUP* is a language that has emerged to improve expressiveness when encoding any CSP to SAT by allowing:

- Define int or bool parameters
- Use *forall* structures to loop over parameters.
- Use *if* structures for conditionally apply constraints.
- Generate clause lists by using comprehension lists.
- Translate automatically any allowed formula to CNF.
- Customize the output when the model is satisfable.
- Produce a clear CNF encoding resulting of the individual conjunction of CNFs resulting from translating each particular constraint.
- Be easily extendible to support further constraints implemented in the SMT API.

3.2 *GOS* compiler

The *GOS* compiler allows to use the defined language, *BUP*, to solve *CSP*s. Given a *BUP* model and a *JSON* file with the data of a particular instance at hand, *GOS* compiler makes the translation to *SAT* and gives the option to print the resulting formula in a standard format, *DIMACS*, or to obtain a solution by using *MiniSAT SAT*-solver.

There are two ways of using *GOS* compiler:

- By building the project using *CMake*
- By using the published online version

3.3 Result example

This section shows an example of modelling the *Sudoku* problem using *GOS*.
The **model file** will be written using the *BUP* language and it would define:

- Necessary parameters (initially fixed sudoku values, board size,...).
- Variables used to model the problem.
- Constraints applied over those variables (no repeated numbers in each row, column and sub-square, initial values must be respected,...)

- Output format.

The following code is a real example of using *BUP* modelling the Sudoku problem (model file):

```
viewpoint:
    var p[9][9][9];
    param int iniSudoku[9][9];

constraints:
    forall(i in 0..8, j in 0..8){
        EO(p[i][j][_]); // One value per cell
        AMO(p[i][_][j]); // Each value one time per row
        AMO(p[_][i][j]); // Each value one time per column
    };
    //Each value one time per block
    forall(i in [0,3,6], j in [0,3,6], k in 0..8){
        AMK([p[i+l][j+g][k] | l in 0..2, g in 0..2], 1);
    };
    //Initialize input fixed sudoku values.
    forall(i in 0..8, j in 0..8){
        if(iniSudoku[i][j] != 0){
            p[i][j][iniSudoku[i][j]-1];
        };
    };
output:
    "Sudoku solution: \n";
    [ k+1 ++ " " ++ ((j+1) % 3 == 0 ? " " : "") ++ (j==8 ? (i+1) % 3 == 0 ? "\n↩
    ↪ \n": "\n" : "") | i in 0..8, j in 0..8, k in 0..8 where p[i][j][k]];
```

The **parameters file** would give the values to the required model parameters. The following listing is an example of sudoku instance (parameters file):

```
{
    "iniSudoku" : [
        [8, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 3, 6, 0, 0, 0, 0, 0],
        [0, 7, 0, 0, 9, 0, 2, 0, 0],
        [0, 5, 0, 0, 0, 7, 0, 0, 0],
        [0, 0, 0, 0, 4, 5, 7, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 3, 0],
        [0, 0, 1, 0, 0, 0, 0, 6, 8],
        [0, 0, 8, 5, 0, 0, 0, 1, 0],
        [0, 9, 0, 0, 0, 0, 4, 0, 0]
    ]
}
```

GOS compiler will process these data and, by using the *L∧P API*[1], will generate the propositional formula as a result of applying the defined constraints over the model variables.

[1] L∧P is a research group of the University of Girona that has developed an API to manage *Satisfiability Modulo Theories (SMT)* formulas (an extension of *SAT*). This API also integrates different solvers (among them SAT specific solvers).

Once the formula is generated, the compiler can print the formula in a standard format, *DIMACS*, or use the solver to get a solution.
Given the model and the sudoku instance previously defined, *GOS* generates the following Boolean Formula, expressed in a standard format, *DIMACS*. The user could decide whether print the formula or solve the problem.

```
p cnf 4536 11361
-1 730 0
-2 730 0
-1 -2 731 0
1 -731 0
2 -731 0
1 2 -730 0
...
```

In this example it is shown a chunk of the whole output formula, that consists on 4536 variables and 11361 clauses.

4 Conclusions

The objectives of this project have been accomplished by defining the language *BUP* and implementing its compiler *GOS*. *BUP* allows to model any *CSP* to *SAT* using simple and user-friendly declarative expressions and takes a step forward on the path to defining object-oriented declarative languages, by allowing to create structs, called *entities*.

The tools developed in this project will assist the L∧P group in their current research on SAT encodings. Moreover, these tools will serve for teaching purposes in the *Declarative Programming* subject at the *Univerisity of Girona*.

GOS compiler has also been made available to the research community by creating a GitHub repository and also creating a web tool that will eventually be available.

References

Ansótegui, C., Bofill, M., Palahí, M., Suy, J., & Villaret, M. (2013, nov). Solving weighted CSPs with meta-constraints by reformulation into satisfiability modulo theories. *Constraints*, *18*(2), 236–268. doi: 10.1007/s10601-012-9131-1

Frisch, A. M., Harvey, W., Jefferson, C., Martínez-Hernández, B., & Miguel, I. (2008). Essence: A constraint language for specifying combinatorial problems. *Constraints*, *13*(3), 268–306.

Stuckey, P. J., Marrioo, K., & Tack, G. (2018). *MiniZinc Handbook Release 2.2.1* (Tech. Rep.).

Zhou, N. F., & Kjellerstrand, H. (2016). The picat-sat compiler. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (Vol. 9585, pp. 48–62). Springer Verlag. doi: 10.1007/978-3-319-28228-2_4