Universitat de Girona

A SATISFIABILITY MODULO THEORIES APPROACH TO CONSTRAINT PROGRAMMING

Josep SUY FRANCH

Dipòsit legal: GI. 150-2013
http://hdl.handle.net/10803/98302

PHD THESIS

A Satisfiability Modulo Theories Approach to Constraint Programming

Author:
Josep SUY FRANCH

2012

Programa de Doctorat en Tecnologia

Advisors:
Dr. Miquel BOFILL ARASA
Dr. Mateu VILLARET AUSELLE

Memòria presentada per optar al títol de doctor per la Universitat de Girona

Abstract

Satisfiability Modulo Theories (SMT) is an active research area mainly focused on formal verification of software and hardware. The SMT problem is the problem of determining the satisfiability of ground logical formulas with respect to background theories expressed in classical first-order logic with equality. Examples of theories include linear real or integer arithmetic, arrays, bit vectors, uninterpreted functions, etc., or combinations of them. Modern SMT solvers integrate a Boolean satisfiability (SAT) solver with specialized solvers for a set of literals belonging to each theory.

On the other hand, Constraint Programming (CP) is a programming paradigm devoted to solve Constraint Satisfaction Problems (CSP). In a CSP, relations between variables are stated in the form of constraints. Each constraint restricts the combination of values that a set of variables may take simultaneously. The constraints are stated over specific domains, typically: Booleans, integers, rationals, reals, finite domains, or combinations of them. The problem consists in finding an assignment to the variables that satisfy all constraints. Many CP solving algorithms are based on systematic search, but in the last years there have been many other successful approaches to solve CSPs: SAT, Mixed Integer Linear Programming (MILP), Lazy Clause Generation (Lazy_fd), genetic algorithms, tabu search, ...

In this thesis we focus on solving CSPs using SMT. Essentially, what we do is reformulating CSPs into SMT. There are already promising results in this direction, but there does not exist a generic approach and there are no available tools to extensively explore the adequacy of this approach. We intend to fill this gap with a complete encoding of an standard CP language. We provide extensive performance comparisons between state-of-the-art SMT solvers and most of the available CP solvers on a large collection of problems. The obtained results allow us to conclude that state-of-the-art SMT solvers are a robust tool to solve CSPs.

We tackle not only decisional CSPs, but also Constraint Optimization Problems (COP), where the objective is to find a solution that is optimal with respect to some given objective function. We also address the Weighted Constraint Satisfiability Problem (WCSP), where there are constraints with an associated weight (cost) and where the goal is to minimize the sum of the costs of the unsatisfied constraints. For solving COP

and WCSP we have used SMT in conjunction with appropriated algorithms: search algorithms and UNSAT core based algorithms borrowed from the MaxSAT area.

We have developed the `fzn2smt` system for encoding instances of the MiniZinc standard (CSP and COP) specification language into SMT. With this system we have obtained the golden medal in the *par* division and the silver medal in the *free* division of the MiniZinc challenge 2010, and the silver medal in the same divisions of the MiniZinc challenge 2011.

We have also developed a new specification language, called `Simply`, and its extension `WSimply`, for programming in the three CP paradigms (CSP, COP and WCSP), and a compiler from this language to SMT. `WSimply` provides support for meta-constraints that is, constraints on constraints. Meta-constraints can be very helpful in the modelling process, since they allow us to abstract to a higher level, expressing, e.g., priorities between a set of soft constraints, different levels of preference (multi-objective optimization), etc. As far as we know, `WSimply` is the first declarative CP language allowing to model WCSP instances intensionally and supporting meta-constraints.

Once seen that SMT is a very good approximation for CP, we have tried to test whether algorithms built on top of an SMT solver can have equal or better performance than ad hoc programs designed specifically for a given problem, based on other approaches. We concentrate on scheduling problems. Scheduling problems consist in deciding how to commit resources and time of execution to a set of activities. They are well suited to SMT because have a strong Boolean component and also an important arithmetic component. To test our approach we have chosen the resource-constrained project scheduling problem (RCPSP), and its generalizations RPCPSP/max and MRCPSP, since this is the most widely studied scheduling problem in the literature. We provide extensive performance comparisons between our approach and state-of-the-art RCPSP solvers. We remark that our system outperforms all other approaches described in the literature to solve MRCPSP and it is competitive in RCPSP and RCPSP/max.

Resum

La Satisfactibilitat Mòdul Teories (SMT) és una àrea de recerca activa centrada principalment en la verificació formal de programari i maquinari. Un problema SMT consisteix en determinar la satisfactibilitat de fórmules lògiques sense quantificadors, respecte a teories de fons expressades en lògica clàssica de primer ordre amb igualtat. Alguns exemples d'aquestes teories són: l'aritmètica lineal real o entera, les matrius, els vectors de bits, les funcions no interpretades, ..., o combinacions d'elles. Els solucionadors SMT moderns integren un solucionador de Satisfactibilitat Booleana (SAT) amb solucionadors especialitzats per al conjunt de literals que pertanyen a cada teoria.

D'altra banda, la Programació amb Restriccions (CP) és un paradigma de programació dedicat a resoldre els problemes de satisfacció de restriccions (CSP). En un CSP, les relacions entre les variables s'expressen en forma de restriccions. Cada restricció limita la combinació de valors que poden prendre a la vegada un conjunt de variables. Les restriccions s'expressen a través de dominis específics com ara: Booleans, enters, racionals, reals, dominis finits, o combinacions d'aquests. Aquest problema consisteix en trobar una assignació a les variables que satisfaci totes les restriccions. Molts algorismes de resolució de CSPs es basen en la cerca sistemàtica, tot i que en els últims anys han aparegut molts altres mètodes per resoldre CSPs amb èxit: SAT, Mixed Integer Linear Programming (MILP), Lazy Clause Generation (Lazy_fd), algorismes genètics, recerca tabú, ...

Aquesta tesi es centra en la resolució de CSPs utilitzant SMT. En esència, es reformulen els CSPs a fórmules SMT. Ja existeixen alguns resultats prometedors en aquest sentit, però sense un enfocament genèric. Tampoc existeixen eines disponibles per explorar exhaustivament la idoneïtat d'aquest enfocament. Es preten omplir aquest buit amb una codificació completa d'un llenguatge estàndard de CP. També es proporcionen comparacions exhaustives de rendiment entre els millors solucionadors actuals d'SMT i la majoria dels solucionadors de CP disponibles en una àmplia col·lecció de problemes. Els resultats obtinguts permeten concloure que els millors solucionadors actuals d'SMT són una eina sòlida per a resoldre CSPs.

No només s'aborden els CSP decisionals, sinó també problemes d'optimització de restriccions (COP), on l'objectiu és trobar una solució que sigui òptima respecte a una

funció objectiu donada. També tractem el problema de restriccions de satisfacibilitat amb pesos (WCSP), on hi ha restriccions amb un pes associat (cost) i on l'objectiu és minimitzar la suma dels costos de les restriccions insatisfetes. Per a resoldre COP i WCSP s'ha utilitzat SMT juntament amb els algorismes apropiats: algorismes de cerca i algorismes basats en nuclis d'insatisfactibilitat provinents de l'àrea de MaxSAT.

S'ha desenvolupat el sistema `fzn2smt` per a la codificació de les instàncies del llenguatge d'especificació estandard MiniZinc a SMT. Amb aquest sistema s'ha obtingut la medalla d'or en la divisió "par" i la medalla de plata en la divisió "free" de la MiniZinc Challenge 2010, i la medalla de plata en les mateixes divisions de la MiniZinc Challenge 2011.

També s'ha desenvolupat un nou llenguatge d'especificació, anomenat `Simply`, i la seva extensió `WSimply`, per a la programació dels tres paradigmes de CP (CSP, COP i WCSP), i un compilador d'aquests llenguatges a SMT. `WSimply` dóna suport a meta-restriccions, és a dir, restriccions sobre restriccions. Les meta-restriccions poden ser de gran ajuda en el procés de modelatge, atès que ens permeten abstreure'ns a un nivell superior, per expressar, per exemple, les prioritats entre un conjunt de restriccions amb pes, diferents nivells de preferència (optimització multiobjectiu), etc. Pel que sabem, `WSimply` és el primer llenguatge declaratiu de CP que permet modelar WCSPs intensionalment i alhora donar suport a meta-restriccions.

Un cop comprovat que SMT és una molt bona aproximació per a CP, s'ha tractat de comprovar si els algorismes basats en SMT (amb ajuda d'un solucionador SMT directament o utilitzant-lo com un oracle) poden tenir un rendiment igual o millor que els programes dissenyats específicament per a un problema donat, desenvolupats sobre la base d'altres enfocaments. Ens concentrem en els problemes de programació d'activitats (scheduling). Els problemes de programació d'activitats consisteixen en decidir com assignar recursos i temps d'execució a un conjunt d'activitats. Aquest problemes s'adapten bé a SMT ja que tenen una forta component Booleana i també una component aritmètica important. Per comprovar el nostre enfocament s'ha escollit el Problema del Projecte de Programació amb Recursos Limitats (RCPSP), i les seves generalitzacions RPCPSP/max i MRCPSP, atès que aquest és el problema de programació d'activitats més àmpliament estudiat en la literatura. Es proporcionen comparacions exhaustives de rendiment entre el nostre enfocament i les millors tècniques actuals per resoldre RCPSP. Cal destacar que el nostre sistema supera tots els altres mètodes descrits a la literatura per resoldre MRCPSP i és competitiu en RCPSP i RCPSP/max.

Acknowledgments

I would like to acknowledge the people who in some way have contributed to the success of this work. First of all, I would like to express my sincere gratitude to my advisors, Miquel Bofill and Mateu Villaret, whose support, encouragement and guidance during this time enabled me to develop this thesis.

This thesis is dedicated to my wife Joana and my children Pau and Ferran, thank for their infinite patience, help and unconditional support during these years of work. Thanks also, to my mother Salvadora and a special memory for my father Francesc.

I would also like to express my gratitude to all the members of the Logic and Programming Research Group $(L \wedge P)$ of the Universitat de Girona : Miquel Palahí, Joan Espasa, Marc Massot, also to Carlos Ansótegui of the Universitat de Lleida; the students who have worked with me these last years: David Moreno and Jordi Coma. I also appreciate the help and comments of many colleagues in the department of IMAE-LSI, including: Joan Surrell, Santi Thio, Josep Soler, and Imma Boada.

The work presented in this thesis was partially supported by by the Spanish Ministry of Science and Innovation through the project TIN2008-04547.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation and Objectives

Constraint Satisfaction Problems (CSPs) are decisional problems expressed with constraints. Typically, CSPs are computationally intractable (NP-hard) combinatorial problems. However, the techniques and algorithms developed in recent decades show that many instances can be solved in a reasonable time, although there will always be instances with very long solving time. Constraint Programming (CP) is the programming paradigm devoted to solve CSPs.

There are a lot of approaches to solve CSPs. The classical approach is to use systematic search algorithms [DF98, Gas79, HE80] with consistency techniques [Kum92, Mac77, MH86] and propagation (Forward Checking, Look Ahead). But there are many other approaches including, among others, genetic algorithms, and Mixed Integer Linear Programming (MILP), consisting in mapping a CSP to a mixed integer linear program and applying local search techniques for moving from solution to solution in the space of candidate solutions.

An approach that has achieved great results in the last years is reducing the CSP into the Boolean Satisfiability Problem (SAT), and then finding a solution with an state-of-the-art SAT solver [Wal00, CMP06, TTKB09]. Nowadays, this is considered one of more powerful generic CSP solving approaches. It has proven to be highly competitive in a variety of problems [MMZ$^+$01, VB01, ZLS04, LMS06, Kau06, KSHK07]. However, this approach is a bit unfriendly, since encoding domains of variables and constraints into SAT is tedious and lengthy.

On the other hand, in recent years there have been important developments in Satisfiability Modulo Theory (SMT) solvers. An SMT formula is a generalization of a Boolean formula in which some propositional variables have been replaced by predicates with pre-

defined interpretations from background theories, such as linear integer arithmetic, arrays, bit vectors, uninterpreted functions, etc., or combinations of them. The goal of an SMT solver is to check if the Boolean formula is satisfiable while the truth assignment chosen for the predicates is consistent with respect to the background theory. Modern SMT solvers integrate a SAT solver with specialized solvers for a set of literals belonging to each theory. These solvers were originally developed for solving problems of software and hardware verification, that typically are small but computationally hard. Although most SMT solvers are restricted to decidable quantifier free fragments of their logics, this suffices for many applications.

The objectives of the thesis are the following:

1. The first objective is to show that SMT can be a good generic solving approach for CSP. In fact, there are already promising results in the direction of adapting SMT techniques for solving some particular CSPs [NO06, NORCR07]. This approach has the advantage of combining the power of SAT solvers and of theory solvers to get the best of both worlds. Another aspect to consider is that, thanks to the use of theories with high expressiveness, encodings can be simpler and more compact than in plain SAT.

   With the aim of bringing a bit more of SMT technology to CP, we have started by developing a new language and system for modelling and solving CSPs with SMT, called `Simply`. Then, we have developed `fzn2smt`, a systematic and general encoding of the MiniZinc standard CSP specification language into SMT.

2. The second objective is to prove that using an SMT solver in conjunction with appropriate algorithms can be a robust approach for optimization variants of CSP.

   Constraint Optimization Problems (COP) are an optimization variant of CSP where the objective is to find a solution that optimizes some given objective function.

   Weighted CSP (WCSP) is another variant used for modelling over-constrained problems, in which (soft) constraints may have an associated weight (cost of falsification) and the goal is to find a solution that minimizes the sum of costs of unsatisfied constraints. The equivalent problem in SAT is called Weighted MaxSAT. One of the most efficient solving techniques for this last problem is the use of UNSAT core based algorithms. Therefore, we think that it is worthily to adapt those UNSAT core based algorithms for SMT.

3. The third objective is to develop a system supporting meta-constraints, allowing the user to model Weighted CSP intensionally, and to solve them using SMT.

   This is motivated mainly by the aim of developing an efficient and at the same time user friendly programming system. Although in over-constrained problems the preferences on which constraints to violate can be modelled by attaching a weight

to each constraint, we may wish to go further at the specification level by allowing the user to express her preferences more easily, or even to express more complex preferences. In [PRB00], a set of constraints on soft constraints, called meta-constraints, was introduced. Meta-constraints can be very helpful in the modelling process, since they allow us to abstract to a higher level, expressing, e.g., priorities between a set of soft constraints, different levels of preference (multi-objective optimization), etc. Hence, the inclusion of meta-constraints increases the capability to easily model several real-world problems. Nevertheless, as far as we know, there does not exist any system supporting the intensional modelling of Weighted CSP.

After having successfully accomplished the three previous objectives, and having seen that SMT is competitive with generic CP tools, we wondered whether algorithms built on top on an SMT solver could have equal or better performance than ad hoc programs based on other approaches, at least for some families of problems. Since SMT exhibited very good performance for scheduling problems, we considered this family. This led to the following last objective.

4. The fourth objective is to provide an SMT based system being competitive with state-of-the-art methods for scheduling problems.

   Scheduling problems consist in deciding how to commit resources and time of execution to a set of tasks. On the one hand, these type of problems have a very important component of integer arithmetic: precedence delays between activities and sums of resource consumptions to verify that the activities running at a certain time do not exceed the specified resource availability. On the other hand, they have a strong Boolean component expressing incompatibilities and precedences between activities. Due to this combination of arithmetic and Boolean constraints, we believe that scheduling problems are ideally suited to the characteristics of SMT solvers.

   To this end we consider the resource-constrained project scheduling problem (RCPSP), which is the scheduling problem more widely discussed in the literature, and its generalizations RPCPSP/max and MRCPSP.

The four objectives presented can be summarized in one conclusion: Satisfiability Modulo Theories can be an efficient and competitive approach to Constraint Programming.

1.2 Contributions

The contributions of this thesis can be summarized as follows:

1. **Encoding CSP into SMT**. We have developed some relatively general and systematic ways of translating CSPs into SMT. We have developed two systems:

   - `Simply`, with its own language, which allows to model CSPs and to solve them via translation to SMT, and
   - `fzn2smt`, that solves instances of the MINIZINC standard CSP language using also SMT.

     We remark that we provide support no only for constraints on integer variables but also for other MINIZINC supported constraints on reals and Booleans, and on vectors and sets of basic types.

2. **Solving COP with SMT**. We have implemented a search procedure to deal with optimization in SMT. This procedure successively calls the SMT solver with slightly modified versions of the instance to optimize. These modifications simply narrow the domain of the variable to be optimized with the addition of constraints. We have implemented three possible bounding strategies: linear, dichotomic and hybrid. Particular strategies following this line have already been applied with great success on particular problems [NO06].

3. **Study of the impact of the Boolean component**. We have studied the impact of the Boolean component of the instances in the performance of our encodings. We have statistically concluded than the greater is the Boolean component, the better is the performance of SMT with respect to other systems.

4. **Empirical proof of the good performance of SMT for solving CSP and COP**. We have performed tests showing that the translation to SMT is a good approximation for solving CSP and COP. We have participated in the *MiniZinc Challenge 2010* and *MiniZinc Challenge 2011* international CSP competitions achieving always medal positions.

5. **Solving intensional WCSP with SMT**. The most friendly way of specifying WCSP is using intensional representations of the constraints. For this reason, we have implemented `WSimply`, a language and system supporting intensional WCSP specifications. We have developed an encoding of these problems in SMT, with the aim of getting a user-friendly and efficient programming system. To solve these problems we use several approaches, including UNSAT core-based algorithms.

6. **Empirical proof of the good performance of SMT for solving WCSP**. We have performed tests showing that encoding WCSP instances in SMT is a robust approximation for solving WCSP. For the sake of completeness we have also implemented an algorithm to encode extensional WCSP in SMT and we have also tested its performance.

7. **Supporting meta-constraints**. We have built a system that supports the meta-constraints proposed in the literature and some other new ones. We solve the meta-constraints by reformulation into SMT, achieving good performance.

8. **Solving scheduling problems with SMT**. We have tested the use of SMT for scheduling, obtaining very good performance. We have developed ad hoc programs for the RCPSP and its generalizations.

   (a) **Four formulations for the RCPSP**. We have implemented four different encodings of the RCPSP in SMT. There exist four well-known formulations for the RCPSP in MILP: time, task, flow and event based. We have translated, modified and improved these formulations to work properly in SMT.

   (b) **New event based formulation**. We have developed a completely new event-based formulation for the RCPSP. This formulation allows us to obtain a much better performance than the direct adaptation of the MILP event-based one.

   (c) **Preprocessing and solving improvements**. We have found that a good preprocessing and an appropriate algorithm on top of the SMT solver is very important to obtain good performance on the RCPSP.

   (d) **RPCPS/max formulation**. We propose an SMT encoding for the RCPSP/max with good performance.

   (e) **MRCPSP formulation**. We propose an SMT encoding for the MRCPSP, with better performance than state-of-the-art solvers on this problem.

   (f) **Empirical proof of the good performance of SMT for solving scheduling problems**. We have performed tests showing that encoding the RCPSP in SMT is a good solving approximation for this problem and its generalizations.

1.3 Publications

Most of the results presented in this thesis have already been published (or have been accepted for publication) in journals and conference proceedings. The list of publications, in chronological order, is the following:

- Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *`Simply`: a Compiler from a CSP Modeling Language to the SMT-LIB Format*. Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation (ModRef), co-located with CP 2009, Lisboa, Portugal, pages 30–44, 2009.

- Miquel Bofill, Josep Suy, and Mateu Villaret, *A System for Solving Constraint Satisfaction Problems with SMT*. Proceedings of the 13th International Conference on

Theory and Applications of Satisfiability Testing (SAT 2010), Edinburgh, United Kingdom, pages 300-305, Springer LNCS, vol. 6175, 2010.

- Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem*. Ninth Symposium on Abstraction Reformulation and Approximation (SARA 2011), Cardona, Spain, pages 2-9, AAAI, 2011.

- Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *A Proposal for Solving Weighted CSPs with SMT*. Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation (ModRef), co-located with CP 2011, Perugia, Italy, pages 5-19, 2011

- Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *W-MiniZinc: A Proposal for Modeling Weighted CSPs with MiniZinc*. Proceedings of the first Minizinc workshop (MZN), co-located with CP 2011, Perugia, Italy, 2011

- Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Solving constraint satisfaction problems with SAT modulo theories*. Constraints journal, vol. 17, number 3, pages 273-303, 2012.

- Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Solving Weighted CSPs with Meta-Constraints by Reformulation into Satisfiability Modulo Theories*. Constraints journal, accepted with minor changes.

1.4 Outline of the Thesis

This section briefly describes the contents of each chapter of the thesis.

Chapters 2, 3 and 4 provide the main required background.

- Chapter 2: **Constraint Programming.** We present the Constraint Satisfaction Problem (CSP). First, we introduce some basic concepts commonly used in Constraint Programming (CP). Then we present the most widely used methods for solving CSPs: systematic search and consistency techniques, as well as the propagation techniques integrated in search algorithms. Then, we provide a brief description of the branch and bound algorithm used to solve constraint optimization problems. We also review other solving approaches historically used to solve CSPs. Finally, we describe over-constrained problems and their associated formalism called Weighted CSP.

- Chapter 3: **Satisfiability.** We provide an overview of the most relevant techniques for solving SAT, the problem of deciding the satisfiability of Boolean formulas. First, we introduce some basic concepts commonly used in SAT. Second, we present the resolution method, which applies an inference rule that provides a refutation complete inference system. Third, we describe the DP algorithm, the first effective method for producing resolution refutations. Fourth, we present the DLL procedure, implemented in the majority of state-of-the-art complete SAT algorithms. We end the SAT solving part by introducing abstract DPLL.

  Next, we introduce the MaxSAT and Weighted MaxSAT formalisms, and review the main solving techniques that have proved to be useful in terms of performance for these paradigms.

  Finally, we present an overview of the existing encodings from CSP into SAT. We review the different ways of encoding CSP variables into SAT and describe, among others, the direct and the support encoding, which are the most frequently used encodings for constraints.

- Chapter 4: **Satisfiability Modulo Theories.** We present the Satisfiability Modulo Theories (SMT) problem. First, we introduce some basic concepts commonly used in Satisfiability Modulo Theories, and describe the so called lazy approach for SMT. We also present some of the more relevant theories (IDL, LIA, EUF, arrays, bit vectors, ...). Then we present the MaxSMT and Weighted SMT Problems and the techniques used to solve these optimization variants. Finally, we introduce Lazy_fd, which is an approach for solving CSP very similar to that of SMT solvers.

Chapters 5, 6, 7 and 8 are the core of the thesis and contain all the contributions.

- Chapter 5: **Encoding CSP into SMT.** We begin by briefly describing the state-of-the-art of encoding CSP into SAT and SMT. Then we describe the first language and system that we have developed, called `Simply`, for modelling and solving CSPs with SMT. Next we present MiniZinc, a high-level standard CSP specification language, and FlatZinc, an "intermediate" specification language obtained after flattening MiniZinc. Then, we present a new and complete system for encoding CSP into SMT, called `fzn2smt`, which essentially consists in encoding FlatZinc instances into SMT. We also describe some solutions to solve optimization problems within this system. To conclude we provide extensive benchmarking and a discussion of the impact of the Boolean component of the instances in the solving performance when using SMT.

- Chapter 6: **Weighted CSP and Meta-Constraints.** In this chapter we introduce a new language for modelling Weighted CSP and a system for encoding Weighted CSP into Weighted SMT. We first introduce `WSimply`, a declarative intensional

Weighted CSP specification language with support for meta-constraints. We carefully specify the supported meta-constraints and illustrate their usefulness. Then we show which solving strategies can be followed, either via optimization or via specialized algorithms for Weighted SMT developed in our works for the first time. Some of these algorithms are adaptations of core-based algorithms for MaxSAT. Then, we provide a large number of experiments showing the possibilities of this system. Finally, we describe how do we deal with extensional Weighted CSP and the reasonably good results obtained.

- Chapter 7: **Scheduling.** In Chapters 5 and 6 we have shown that SMT is a good choice for solving CSP. Moreover, SMT is really good in solving problems with a strong Boolean component with integer arithmetic, such as scheduling problems. In this chapter we present some encodings for the Resource Constrained Project Scheduling Problem (RCPSP) into SMT. RCPSP is the scheduling problem more widely discussed in the literature. First we describe the state-of-the-art in RCPSP, where in recent years there have been a lot of approaches (CP, SAT, MILP, ...) but not SMT. Second, we formally present this problem. Then we present our approach with preprocessing and four different encodings with many variants. Finally, we provide a large number of experiments showing that SMT is not only competitive with generic CP tools, but also that algorithms built on top of an SMT solver can have equal or better performance than ad hoc programs based on other approaches.

- Chapter 8: **Other Scheduling Problems.** We present two generalizations of the RCPSP with even more Boolean component. First we present a system (preprocessing, encoding and solving) to solve the RCPSP/max obtaining good results. Second we present a very good system to solve the multimode RCPSP (MRCPSP), that outperforms all current existing approaches.

- Chapter 9: **Conclusions and Future work.** In this last chapter we summarize the main contributions of the thesis, and propose future research directions to work on.

Chapter 2

Constraint Programming

Constraint Programming (CP) is a programming paradigm where the relations between the variables are stated in the form of constraints. Each constraint restricts the combination of values that a set of variables may take simultaneously. Constraints are additive, i.e., the order of the constraints does not matter, as all that matters is the conjunction of all specified constraints. This makes constraint programming a kind of declarative programming.

The constraints used in constraint programming are typically over specific domains. Some domains for constraint programming are: Boolean, integer, rational, reals, finite domains, mixed domains (involving two or more domains). Finite domains are one of the most successful domains of constraint programming, in fact, constraint programming is often identified with constraint programming over finite domains.

In this chapter we present constraint programming and Constraint Satisfaction Problems (CSP). We provide an overview of the solving techniques frequently used in the classical CSP solvers: systematic search, consistency techniques, propagation and Constraint Optimization (COP). We also present global constraints, which are widely used to improve the performance of CSP solvers. We also introduce other CSP solving methods, some of which are described in more detail in subsequent chapters. Finally we describe variants of CSP such as MaxCSP and Weighted CSP.

This chapter is partially based on [Bar05, RBW06].

2.1 Constraint Satisfaction Problems

Constraint programming is devoted to solve constraint satisfaction problems, which are formally defined as follows:

Definition 2.1.1 *A* constraint satisfaction problem (CSP) *is defined as a triple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$*, where* $\mathcal{X} = \{X_1, \ldots, X_n\}$ *is a set of variables,* $\mathcal{D} = \{D(X_1), \ldots, D(X_n)\}$ *is a set of domains containing the values that each variable may take, and* $\mathcal{C} = \{C_1, \ldots, C_m\}$ *is a set of constraints. Each constraint* $C_i = \langle S_i, R_i \rangle$ *is defined as a relation* R_i *over a subset of variables* $S_i = \{X_{i_1}, \ldots, X_{i_k}\}$*, called the* constraint scope*. A relation* R_i *may be represented* intensionally*, in terms of an expression that states the relationship that must hold amongst the assignments to the variables it constrains, or it may be represented* extensionally*, as a subset of the Cartesian product* $D(X_{i_1}) \times \cdots \times D(X_{i_k})$ *(tuples) which represents the allowed assignments (*good tuples*) or the disallowed assignments (*no-good tuples*). Constraints of arity* n *are called* n*-ary.*

Definition 2.1.2 *A* partial assignment v *for a CSP* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *is a mapping that assigns to every variable* $X_i \in \mathcal{Y}$ *an element* $v(X_i) \in D(X_i)$*, where* $\mathcal{Y} \subseteq \mathcal{X}$ *is the domain of* v*, denoted* $domain(v)$*; when* $\mathcal{Y} = \mathcal{X}$ *it is simply an* assignment*. A (partial) assignment* v *satisfies a constraint* $\langle \{X_{i_1}, \ldots, X_{i_k}\}, R_i \rangle \in \mathcal{C}$ *if and only if* $\langle v(X_{i_1}), \ldots, v(X_{i_k}) \rangle \in R_i$ *(assuming that* R_i *is intensional or represents good tuples).*

A solution to a CSP *is an assignment where every constraint is satisfied.*

A (partial) assignment v *is* consistent *iff satisfies all the constraints* $\langle S_i, R_i \rangle \in \mathcal{C}$ *such that* $S_i \subseteq domain(v)$*. Otherwise we say that it is inconsistent or there is a conflict.*

Definition 2.1.3 *A* label (X, val) *is a variable-value pair which represents the assignment of value* val *to variable* X*.*

Example 1 *Consider* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *where* $\mathcal{X} = \{X_1, X_2, X_3\}, D(X_1) = D(X_2) = D(X_3) = \{1, 2, 3, 4\}$ *and* $\mathcal{C} = \{X_1 < X_2, X_2 < X_3, C_3(X_1, X_2, X_3)\}$ *with* $C_3(X_1, X_2, X_3) = \{(1, 2, 3), (1, 3, 4), (2, 1, 2), (2, 3, 4)\}$*. In this example we have two intensional constraints,* $X_1 < X_2$ *and* $X_2 < X_3$*, and one extensional constraint* C_3 *where the tuples are goods. The first two constraints are binary (arity* 2*), and the third one is* 3*-ary. The set of labels* $\{(X_1, 1), (X_2, 2), (X_3, 3)\}$ *denotes an assignment. This assignment satisfies all the constraints. Therefore, it is consistent and constitutes a solution to the problem.*

CSPs are combinatorial problems that can be solved by search but systematic search is usually unfeasible in practice. So, one of the main research topics in the area of constraint satisfaction consists in finding efficient constraint solving algorithms.

2.2 Systematic Search

Many CSP solving algorithms do systematic search through the possible assignments. Such algorithms are guaranteed to find a solution, if one exists, or to prove that the problem is insoluble. Thus systematic search algorithms are sound and complete. The main

disadvantage of these algorithms is that they may take a very long time. There are two main classes of systematic search algorithms:

1. Algorithms that search the space for complete assignments, i.e., assignments of all variables, till they find a complete assignment that satisfies all the constraints.

2. Algorithms that extend a partial consistent assignment to a complete assignment that satisfies all the constraints.

In general, the tasks posed in the constraint satisfaction problem paradigm and in particular the systematic search are computationally intractable (NP-hard). However, the techniques and algorithms developed in recent decades show that many instances can be solved in a reasonable time, but we will always have instances with very high time resolution.

In this section we present basic representatives of both classes: generate and test, and backtracking. Although these algorithms are simple and inefficient, they are very important because they make the foundation of other algorithms that exploit more sophisticated techniques like propagation or local search.

2.2.1 Generate and Test

The generate and test (GT) algorithm generates some complete assignment and then it tests whether this assignment satisfies all the constraints. If the test fails, then it considers another complete assignment. The algorithm stops as soon as a complete assignment satisfying all the constraints is found (it is a solution) or all complete assignments have been generated without finding the solution (the problem is unsolvable).

Since the GT algorithm systematically searches the space of complete assignments, i.e., it considers each possible combination of variable assignments, the number of combinations considered by this method is equal to the size of the Cartesian product of all variable domains.

The pure generate-and-test approach is not efficient because it generates many wrong assignments of values to variables which are rejected in the testing phase. In addition, the generator ignores the reason of the inconsistency and it generates other assignments with the same inconsistency.

In Section 2.7 we briefly describe the local search method, a more clever approximation to the search algorithms based on complete assignments.

Algorithm 1 Generate and test algorithm

Input: $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: CSP instance
Output: Existence of solution
  for all v: assignment **do**
    if consistent($v, \mathcal{C}$) **then**
      return $true$
    end if
  end for
  return $false$

2.2.2 Backtracking

The backtracking (BT) algorithm incrementally attempts to extend a partial assignment, that specifies consistent values for some of the variables, towards a complete and consistent assignment. This extension is done by repeatedly choosing a value for a not yet assigned variable, until a consistent value is found according to the current partial assignment. This algorithm is the most common algorithm for systematic search [DF98].

In the BT method, variables are instantiated sequentially and as soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial assignment violates any of the constraints, a backtracking step is performed by reconsidering the assignment to the most recently instantiated variable that still has available alternatives. Clearly, whenever a partial assignment violates a constraint, backtracking is able to prune a subspace from the Cartesian product of all variable domains. Consequently, backtracking is strictly better than generate-and-test. However, its running time complexity for most nontrivial problems is still exponential.

The basic form of backtracking, called chronological backtracking, is shown in Algorithm 2. If this algorithm discovers an inconsistency then it always backtracks to the last (according to the chronological order) decision.

The backtracking algorithm has many drawbacks, among which we can highlight:

1. **Thrashing.** Backtracking does not necessarily identify the real reason of the conflict (inconsistency). Therefore, search in different parts of the search space keeps failing for the same reason. Thrashing can be avoided with backjumping [Gas79].

2. **Redundant work.** Even if the conflict reasons are identified during backjumping, they are not remembered for immediate detection of the same conflict in subsequent computations. Backchecking and backmarking [HE80] solve this problem.

3. **Late conflict detection.** The backtracking algorithm still detects the conflict too late as it is not able to do it before the conflict really occurs, i.e., after assign-

Algorithm 2 Backtracking algorithm

Input: $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: CSP instance
Output: Existence of solution

$Nonassign \leftarrow \mathcal{X}$
$cur \leftarrow 1$
$v \leftarrow \emptyset$
$X_{cur} \leftarrow SelectVariable(Nonassign)$
$D_{cur} \leftarrow D(X_{cur})$
$Nonassign \leftarrow Nonassign \setminus \{X_{cur}\}$
while $cur > 0$ **do**
  if $D_{cur} \neq \emptyset$ **then**
    $val_{cur} = SelectValue(D'_{cur})$
    $D_{cur} \leftarrow D_{cur} \setminus \{val_{cur}\}$
    $v \leftarrow v \cup \{(X_{cur}, val_{cur})\}$
    if consistent$(v, \mathcal{C})$ **then**
      if $Nonassign \neq \emptyset$ **then**
        $cur \leftarrow cur + 1$
        $X_{cur} \leftarrow SelectVariable(Nonassign)$
        $D_{cur} \leftarrow D(X_{cur})$
        $Nonassign \leftarrow Nonassign \setminus \{X_{cur}\}$
      else
        return *true*
      end if
    else
      $v \leftarrow v \setminus \{(X_{cur}, val_{cur})\}$
    end if
  else
    $Nonassign \leftarrow Nonassign \cup \{X_{cur}\}$
    $cur \leftarrow cur - 1$
    if $cur > 0$ **then**
      $v \leftarrow v \setminus \{(X_{cur}, val_{cur})\}$
    end if
  end if
end while
return *false*

ing the values to all the variables of the conflicting constraint. This drawback can be avoided by applying consistency techniques to anticipate the possible conflicts. These techniques are explained in the next section.

Backjumping

The control of backjumping is exactly the same as backtracking, except when the backtrack takes place. Both algorithms pick one variable at a time and look for a value for this variable making sure that the new assignment is compatible with values committed to so far. However, if backjumping finds an inconsistency and all the values in the domain are explored, it analyses the situation in order to identify the source of inconsistency using the violated constraints as a guidance to find out the conflicting variable. Once the analysis is made the backjumping algorithm backtracks to the most recent conflicting variable. Notice than the backtraking algorithm always return to the immediate past variable.

Backchecking and Backmarking

The backchecking and its descendent backmarking are useful algorithms for reducing the number of compatibility checks. If backchecking finds that some label (Y, b) is incompatible with some recent label (X, a) then it remembers this incompatibility. As long as (X, a) is still committed to, (Y, b) will not be considered again.

Backmarking is an improvement over backchecking that avoids some redundant constraint checking by remembering for every label the incompatible recent labels. This avoids the duplication of controls which have already been successfully done.

2.3 Consistency Techniques

Consistency techniques were introduced to improve the efficiency of the search algorithms [Kum92]. The number of possible combinations that should be explored by these search algorithms can be huge, while only very few assignments are consistent. Consistency techniques effectively rule out many inconsistent assignments at a very early stage, and thus cut short the search for consistent assignment. Typically this prunning is done by removing values from the domain of the variables. Consistency techniques are sound, in the sense that when the domain of a variable becomes empty, i.e., the consistency algorithm fails achieving consistency, we can ensure that the CSP has no solution. However, it is not complete, since even achieving consistency, the CSP does not necessarily have a solution.

Example 2 *Consider the CSP instance* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *with* $\mathcal{X} = \{A, B\}$, $D(A) = \{4, \ldots, 9\}$, $D(B) = \{1, \ldots, 6\}$ *and* $\mathcal{C} = \{B > A\}$. *The consistency techniques can make the domains smaller according to the constraints:* $D(A) = \{4, \ldots, 5\}$, $D(B) = \{5, \ldots, 6\}$. *Then, for each possible value in* $D(A)$*, it is possible to find a consistent value for* B *in* $D(B)$*, and vice versa. Note however that this reduction does not remove all inconsistent pairs of labels: for instance,* $\{(A, 5), (B, 5)\}$ *is still a feasible assignment according to the domains of the variables.*

Every CSP can be converted into an equivalent binary CSP, where all constraints are binary [Fre78, DP89, Dec90, RPD90]. Binary CSPs can be represented by a constraint graph. In these graphs nodes are labeled with variables and edges connect pairs of variables being the scope of some constraint. Unary constraints can be represented by cyclic edges. There are many techniques to look for consistency in the constraint graphs and prune the search space.

2.3.1 Node Consistency

The simplest consistency technique is referred to as *node consistency* (NC).

Definition 2.3.1 *The node representing a variable* X *in a constraint graph is* node consistent *if, and only if, for every value* x *in the current domain* $D(X)$*, each unary constraint on* X *is satisfied. A CSP is* node consistent *if and only if all variables are node consistent, i.e., for all variables, all values in their domain satisfy the constraints on that variable.*

2.3.2 Arc Consistency

Since, in the constraint graph, arcs correspond to binary constraints, the consistency of binary constraints is referred to as arc consistency.

Definition 2.3.2 *An arc* (X, Y) *of the constraint graph is* arc consistent *if, and only if, for every value* x *in the current domain* $D(X)$ *satisfying the constraints on* X*, there is some value* y *in the domain* $D(Y)$ *such that the assignment* $\{(X, x), (Y, y)\}$ *is permitted by the binary constraint between* X *and* Y*. Note that the concept of arc-consistency is directional, i.e., if an arc* (X, Y) *is consistent, then it does not automatically mean that* (Y, X) *is also consistent. A CSP is* arc consistent *if and only if every arc in its constraint graph is arc consistent.*

An arc (X, Y) can be made consistent by simply deleting those values from the domain $D(X)$ for which there does not exist a corresponding value in the domain $D(Y)$

such that the binary constraint between X and Y is satisfied. This procedure does not eliminate any solution of the original CSP.

To make a CSP arc consistent, i.e., to make every arc of the corresponding constraint graph arc consistent, it is not sufficient to execute the consistency procedure for each arc just once. One execution of the consistency procedure may reduce the domain of some variable X; then each previously revised arc (Y, X) has to be revised again, because some of the members of the domain $D(Y)$ may no longer be compatible with any remaining members of the pruned domain $D(X)$. The easiest way to establish arc consistency is to apply the consistecy procedure to all arcs repeatedly till no domain of any variable changes. There are several well-known algorithms to achieve arc consistency:

1. **AC-1** [Mac77]. This algorithm is not very efficient since it repeats the algorithm of consistency on all arcs if there has been some domain change in the previous iteration.

2. **AC-3** [Mac77]. This algorithm is a variation of AC-1 where the consistency test is repeated only for those arcs that are possibly affected by a previous revision (see Algorithm 3).

Algorithm 3 AC-3

Input: $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: CSP instance, G : constraint graph
Output: Arc consistency of G

```
Q ← {(X_i, X_j) ∈ G, i ≠ j}
while not Q = ∅ do
  (X_k, X_m) = SelectArcAndDelete(Q)
  if Consistence(⟨X, D, C⟩, (X_k, X_m)) then
    for all (X_i, X_k) ∈ G, i ≠ k do
      Q ← Q ∪ {(X_i, X_k)}
    end for
  end if
end while
return true
```

Consistence($\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle, (X_k, X_m)$) eliminates the values of the domain $D(X_k)$ for which there is no value for X_k in $D(X_k)$ satisfying the constraint of the arc (X_k, X_m). In the case of removing at least one value of the domain $D(X_k)$, this procedure returns $true$, otherwise it returns $false$.

---

3. **AC-4** [MH86]. This algorithm works with individual pairs of values, using *support sets* for each value. A value is supported if there exists a compatible value in the domain of every other variable. When a value x is removed from $D(X)$, it is not always necessary to examine all the binary constraints (X, Y). More precisely, we

can ignore those values in the domain $D(Y)$ whose support does not only rely on x. In other words, we can ignore the cases where every value in the domain $D(Y)$ is compatible with some value in the domain $D(X)$ other than x. In order to always know the support sets for all variables, the AC-4 algorithm must maintain a complex data structure.

Arc consistency does not guarantee the existence of a solution for arc consistent CSP instances. Figure 2.1 illustrates a CSP instance where all constraints are arc consistent but overall the problem has no solution. Therefore, arc consistency is not enough to eliminate the need for backtracking.

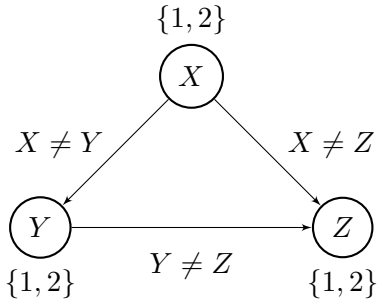


Figure 2.1: Arc consistent CSP without solution.

2.3.3 Other Consistency Techniques

There are stronger consistency techniques that are more powerful than arc consistency with respect to pruning but at the cost of having a higher complexity. It is a crucial issue to find the appropriate compromise between the amount of search space pruned and the time spent in pruning. Among others, there is *path consistency*, which considers triples of variables instead of tuples, and the k-consistency generalization, that considers k-tuples.

2.4 Constraint Propagation

In this section we show the result of joining some of the concepts explained in the two previous sections: systematic search (backtracking and backjumping) and consistency techniques (arc consistency). As skeleton we use a simple backtracking algorithm that incrementally instantiates variables, extending a partial assignment that specifies consistent values for some of the variables, towards a complete assignment. In order to reduce the search space, some consistency techniques are applied to the constraint graph after

assigning a value to a variable. Depending on the consistency technique applied, we get different constraint satisfaction algorithms.

Notice that consistency techniques are polynomial, as opposed to systematic search, which is non-polynomial. Thus polynomial computation is performed as soon as possible, and search is used only when there is no more propagation to be done.

2.4.1 Forward Checking

Forward checking is the easiest way to prevent future conflicts. Instead of performing arc consistency between instantiated variables, forward checking performs arc consistency between pairs of a not-yet instantiated variable and an instantiated one. Therefore, it maintains the invariant that for every uninstantiated variable there exists at least one value in its domain which is compatible with the values of the already instantiated variables.

The forward checking algorithm is based on the following idea. When a value is assigned to the current variable, any value in the domain of a "future" variable which conflicts with this assignment is (temporarily) removed from the domain. If the domain of a future variable becomes empty, then it is known immediately that the current partial assignment is inconsistent. Consequently, forward checking allows branches of the search tree that will lead to a failure to be pruned earlier than with chronological backtracking. See Algorithm 4.

Algorithm 4 AC for Forward Checking

Input: $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: CSP instance, G : constraint graph, X_c:variable
Output: forward checking induced by the change in the domain $D(X_c)$

```
Q ← {(X_i, X_c) ∈ G, i ≠ c}
consistent ← true
while not Q = ∅ ∧ consistent do
  (X_k, X_m) = SelectArcAndDelete(Q)
  if Consistence(⟨X, D, C⟩, (X_k, X_m)) then
    consistent ← NotEmpty(D(X_k))
  end if
end while
return consistent
```

2.4.2 Look Ahead

Forward checking performs only the checks of constraints between the current variable and future variables. We can extend this consistency checking to even latter variables

that do not have a direct connection with already instantiated variables. In partial look ahead, it is used directional arc consistency (arc consistency only into the arcs (X_i, X_j) where $i < j$ for a given total order on the variables $\mathcal{X}$) to choose the arcs to check. When considering non-directional arc consistency we talk about full look ahead.

Algorithm 5 AC-3 for Look Ahead

Input: $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: CSP instance, G : constraint graph, X_c:variable
Output: look ahead induced by the change in the domain $D(X_c)$

```
Q ← {(X_i, X_c) ∈ G, i > c}
consistent ← true
while not Q = ∅ ∧ consistent do
  (X_k, X_m) = SelectArcAndDelete(Q)
  if Consistence(⟨X, D, C⟩, (X_k, X_m)) then
    Q ← Q ∪ {(X_i, X_k) ∈ G, i > c, i ≠ m}
    consistent ← NotEmpty(D(X_k))
  end if
end while
return consistent
```

2.4.3 Propagators

So far we have seen the classic propagation algorithms. We can define a more abstract notion of propagator (see [Bes06]), so that it can be used by distinct search based (or non-search based) algorithms.

Definition 2.4.1 *A* propagator f *is a monotonically decreasing function from domains to domains. That is,* $f(\mathcal{D}) \sqsubseteq \mathcal{D}$*, and* $f(\mathcal{D}_1) \sqsubseteq f(\mathcal{D}_2)$ *whenever* $\mathcal{D}_1 \sqsubseteq \mathcal{D}_2$*. If* $\mathcal{D}_1 = \{D_1(X_1), \ldots, D_1(X_n)\}$ *and* $\mathcal{D}_2 = \{D_2(X_1), \ldots, D_2(X_n)\}$*, the operator* $\sqsubseteq$ *means that domain* D_1 *is stronger than domain* D_2*, i.e.,* $\forall i \in \{1 \ldots n\}$, $D_1(X_i) \subseteq D_2(X_i)$*.*

When a propagator f *is associated to a constraint* $C = \langle S, R \rangle$*, it is noted* f_C *and acts as follows: if* $\mathcal{D} = \{D(X_1), \ldots, D(X_n)\}$ *then* $f_C(\mathcal{D}) = \{D'(X_1), \ldots, D'(X_n)\}$*, where* $D'(X_j) = D(X_j)\ \forall X_j \notin S$ *and* $D'(X_j) \subseteq D(X_j)\ \forall X_j \in S$*.*

Typically we are interested on "correct" propagators, i.e., propagators that preserve all the consistent assignments. More formally, a propagator f is correct if for all consistent assignments v of a CSP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, v is also a consistent assignment of $\langle \mathcal{X}, f(\mathcal{D}), \mathcal{C} \rangle$.

Example 3 *For the CSP* $\langle \{X, Y, Z\}, \{D(X) = \{1, 2, 3, 4\}, D(Y) = \{2, 3\}, D(Z) = \{2, 3, 4\}\}, \{C_1 = (X = Y), C_2 = (Y \leq Z)\}\rangle$ *we can provide the two following correct*

propagators:

$$f_{C_1}(\mathcal{D}) = \mathcal{D}' \quad \textit{where} \quad \begin{cases} D'(X) = D(X) \cap D(Y) \\ D'(Y) = D(Y) \cap D(X) \\ D'(Z) = D(Z) \end{cases}$$

$$f_{C_2}(\mathcal{D}) = \mathcal{D}' \quad \textit{where} \quad \begin{cases} D'(X) = D(X) \\ D'(Y) = \{d \in D(Y) \mid d \leq max(D(Z))\} \\ D'(Z) = \{d \in D(Z) \mid d \geq min(D(Y))\} \end{cases}$$

Suppose that a search algorithm assigns the value 3 *to* X*. This means that* $D(X) = \{3\}$*. Then, the propagator* f_{C_1} *can be activated and set* $D(Y) = \{3\}$*. This, in turns, activates the propagator* f_{C_2} *reducing the domain of* Z *to* $D(Z) = \{3, 4\}$*. Propagators are activated until a fix point is reached.*

Consistency algorithms (arc consistency, path consistency, ...) can use propagators to maintain consistency. Depending on the filtering power of these, the propagator can guarantee arc consistency or other type of consistency. It turns out that consistency algorithms can also be considered as propagators.

2.5 Constraint Optimization

In many real-life applications, we do not want to find any solution but a good one. The quality of solutions is usually measured by some application dependent function called *objective function*. Then the goal is to find such an assignment that satisfies all the constraints and minimizes or maximizes the objective function. Such problems are called Constraint Optimization Problems (COP).

A Constraint Optimization Problem (COP) is a CSP where we want to find an optimal, or at least a good solution, given some objective function defined in terms of (some of) the variables. More formally:

Definition 2.5.1 *A* Constraint Optimization Problem (COP) *is defined as a tuple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{O} \rangle$*, where* $\mathcal{X}$*,* $\mathcal{D}$ *and* $\mathcal{C}$ *represent, as in the case of a CSP, the variables, domains and constraints, respectively, and* $\mathcal{O}$ *is an objective function mapping every complete assignment to an integer (or real) value. An optimal solution of a COP is an assignment that minimizes (or maximizes)* $\mathcal{O}$ *and satisfies all the constraints. In some cases finding the optimal solution is very expensive, and we will content ourselves with a suboptimal solution.*

In order to find the optimal solution, we potentially need to explore all the solutions of the CSP and compare their values using the optimization function. Therefore, techniques for finding or generating all solutions are more relevant to COP than techniques for finding a single solution. In general, as in the case of CSP, the COP are computationally intractable (NP-hard). However, everyone tries to find techniques and algorithms that are able to solve many instances in a reasonable time.

The most widely used technique for solving COP is branch-and-bound. This algorithm consist in exploring a tree where each node represents a label and the branching from the root to a node represents a partial assignment. During search, the algorithm keeps the cost of the best solution found so far, which is an upper bound u of the problem best solution. At each node, the algorithm computes a lower bound l of the best solution in the subtree below. If l is higher than or equal to u, the algorithm prunes the subtree below the current node, because the current best solution cannot be improved by extending the current assignment. In this algorithm, the function f is the objective function, which maps every solution (complete labeling of variables satisfying all the constraints) to a numerical value. The task is to find such a solution that is optimal regarding the objective function, i.e., it minimizes or maximizes respectively the objective function. The function h is a heuristic, which maps (partial) assignments to a numeric value that is an estimate of the objective function. More precisely, h applied to some partial assignment is an estimate of best values of the objective function applied to all solutions (complete assignments) that rise by extending this partial assignment. Naturally, the efficiency of the branch and bound method is highly dependent on the availability of a good heuristic. In such a case, the B&B algorithm can prune the search sub-trees where the current value of f exceeds its best known value. Notice that the initial bound in the algorithm is infinity, as we are minimizing. However, if we know the value of an approach to the optimum, then we can set this value as the initial value. Consequently, the algorithm will prune more sub-trees before and will be much more efficient.

There are other methods to get optimization. Some systems use a technique known as Optimistic Partitioning, which is a binary search between the minimum and maximum values of the objective function. The optimistic hypothesis is that these divisions will allow faster convergence towards good solutions than the classical approach.

Another variant is the Russian Doll Search (RDS) method [VLS96, MS01], which belongs to the area of scheduling. The idea is to successively solve growing nested subproblems. To do this we define an ordering for the problem variables. In the i-th subproblem we consider from the $n - i + 1$-th variable to the last (hence, the first subproblem involves only the last variable, while the n-th subproblem involves all variables). By recording the solution of each subproblem, the bound on the objective function can be usually improved faster than if considering the whole problem initially. This provides noticeable punning at early search stages.

Algorithm 6 Branch & Bound

Input: $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$: CSP instance
Output: Optimal solution

$Nonassign \leftarrow \mathcal{X}$
$cur \leftarrow 1$
$v \leftarrow \emptyset$
$X_{cur} \leftarrow SelectVariable(Nonassign)$
$D_{cur} \leftarrow D(X_{cur})$
$Nonassign \leftarrow Nonassign \setminus \{X_{cur}\}$
$Bound \leftarrow infinity$
$Best \leftarrow \emptyset$
while $cur > 0$ **do**
  if $D_{cur} \neq \emptyset$ **then**
    $val_{cur} = SelectValue(D_{cur})$
    $D_{cur} \leftarrow D_{cur} \setminus \{val_{cur}\}$
    $v \leftarrow v \cup \{(x_{cur}, val_{cur})\}$
    if consistent$(v, C) \wedge h(Nonassgin, v) < Bound$ **then**
      if $Nonassign \neq \emptyset$ **then**
        $cur \leftarrow cur + 1$
        $X_{cur} \leftarrow SelectVariable(Nonassign)$
        $D_{cur} \leftarrow D(X_{cur})$
        $Nonassign \leftarrow Nonassign \setminus \{X_{cur}\}$
      else
        $Bound \leftarrow f(v)$
        $Best \leftarrow v$
        $v \leftarrow v \setminus \{(X_{cur}, val_{cur})\}$
      end if
    else
      $v \leftarrow v \setminus \{(X_{cur}, val_{cur})\}$
    end if
  else
    $Nonassign \leftarrow Nonassign \cup \{X_{cur}\}$
    $cur \leftarrow cur - 1$
    if $cur > 0$ **then**
      $v \leftarrow v \setminus \{(X_{cur}, val_{cur})\}$
    end if
  end if
end while
return $Best$

2.6 Global Constraints

A global constraint is a constraint that captures a relation between a non-fixed number of variables. An example is the $alldifferent([X_1, \ldots, X_n])$ constraint, which specifies that the values assigned to the variables $X_1, \ldots, X_n$ must be pairwise distinct. Typically, a global constraint is semantically redundant in the sense that the same relation can be expressed as the conjunction of several simpler constraints. We can define a global constraint as follow:

Definition 2.6.1 *Let $\mathcal{C} = \{C_1, C_2, \ldots, C_n\}$ be a set of constraints. The constraint $C_G = C_1 \wedge C_2 \wedge \ldots \wedge C_n$, i.e., the conjuntion of all the constraints C_1, C_2, …, C_n is a* global constraint*. The set of solutions of $\mathcal{C}$ is equal to the set of solutions of C_G.*

Global constraints are often defined from a set of variables and some prototypes of non-decomposable constraints. For example, the $alldifferent(L_X)$ is defined as the conjunction of the $\neq$ constraints for each pair of variables in L_X.

Global constraints have three main advantages:

- It is more convenient to define one constraint corresponding to a set of constraints than to define independently each constraint of this set.
- Since a global constraint corresponds to a set of constraints, it is possible to deduce some information from the simultaneous presence of constraints.
- Powerful consistency algorithms can be designed because the set of constraints can be taken into account as a whole. Specific consistency algorithms allow one to use Operations Research techniques or graph theory algorithms.

2.7 Other CSP Solving Methods

In recent years there has been a proliferation of alternative methods to traditional search trying, with varying success, to solve CSPs. These new approaches include exact methods, methods that do not guarantee the best solution and even methods that do not guarantee any solution. Among them we can find:

- **Mixed Integer Linear Programming (MILP)**. This method consists on mapping a CSP to a mixed integer linear program:

$$min\{c^T x : Ax \leq b, l \leq x \leq u, x \in \mathbb{R}^n, x_j \in \mathbb{Z}\, \forall j \in I\}$$

where $A \in \mathbb{Q}^{m*n}, c \in \mathbb{Q}^n, b \in \mathbb{Q}^m, l \in (\mathbb{Q} \cup \{-\infty\})^n, u \in (\mathbb{Q} \cup \{\infty\})^n$ and $I \subseteq \mathbb{N} = \{1, \ldots, n\}$. Here $c^T x$ is the objective function, $Ax \leq b$ are the linear constraints, l and u are the lower and upper bounds on the problem variables x, and I is the subset of indices denoting the variables required to be integer. The method uses the simplex algorithm to solve the linear program (LP) relaxation:

$$min\{c^T x : Ax \leq b, l \leq x \leq u, x \in \mathbb{R}^n\}$$

Notice that it is the same problem but with the difference that the integer requirement on the x variables indexed by I has been dropped. The main reason for this relaxation is that thanks to this, the relaxed problem turns to be polynomial.

Using the LP computation as a tool, MILP solvers integrate a variant of the branch-and-bound and the cutting plane algorithms [Gom58] of the general branch-and-cut scheme [PR91] to ensure that the variables indexed by I are integer.

- **Satisfiability**. It consists on encoding CSPs into Boolean satisfiability problems (SAT). This approach is described in detail in Section 3.5.

- **SMT**. It consists on encoding CSPs into SAT modulo theories problems. This approach is part of this thesis contribution and is widely described in Chapters 5 and 6.

- **Lazy Clause Generation**. It is a hybrid technique between finite domain propagation and SAT solving that combines some of the strengths of both. This approach is described in detail in Section 4.6.

- **Local search**. This technique consists on moving from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed. A local search algorithm starts from a solution and then iteratively moves to a neighbor solution. This is only possible if a neighborhood relation is defined on the search space. As an example, the neighborhood of a vertex cover is another vertex cover only differing by one node. Despite the inherent incompleteness, these methods have shown great ability to solve difficult problems.

  One of the local search methods most used today is the tabu search [Glo89, Glo90]. Central to tabu search is a tabu list, which keeps a list of moves that may not be performed. Thus, when generating a list of candidates from a current solution, some neighboring solutions cannot be added to the list. The tabu list serves to ensure that the moves in the list are not reversed thus preventing previous moves from being repeated. Criteria for moves entering the tabu list can be defined in many ways, and similar criteria exist for moves to be off the tabu list.

- **Genetic algorithms** [BFM97]. This approach is based on an analogy with the evolution theory. In these algorithms, a state corresponds to a total assignment;

one derives new states by recombining two parent states using a mating function that produces a new state corresponding to a cross-over of the parent states. The parent states are selected from a pool of states called population. The new states are subjected to small random changes (mutations). This approach does not guarantee that a solution is found although it exists.

2.8 MaxCSP and Weighted CSP

A Constraint Satisfaction Problem (CSP) is a decision problem where the objective is to determine whether there exists an assignment of values to a set of variables which satisfies a given set of constraints. However, many real world instances of CSPs are over-constrained and therefore have no solution. There is a formalism that allows us to express these problems, for which a solution must violate as few restrictions as possible, or minimize the costs or penalties associated with the violated constraints. These frameworks are called MaxCSP and Weighted CSP, respectively. The Weighted CSP also is known as a Cost Function Network (CFN).

Definition 2.8.1 *The* MaxCSP *problem for a CSP is the problem of finding an assignment that minimizes (maximizes) the number of violated (satisfied) constraints.*

Definition 2.8.2 *A* Weighted CSP (WCSP) *is a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X}$ and $\mathcal{D}$ are variables and domains, respectively, as in a CSP. A constraint $C_i \in \mathcal{C}$ is now defined as a pair (S_i, f_i), where $S_i = \{X_{i_1}, \ldots, X_{i_k}\}$ is the constraint scope and $f_i : D(X_{i_1}) \times \cdots \times D(X_{i_k}) \to \mathbb{N}$ is a* cost (weight) function *that maps tuples to its associated weight (a natural number or infinity). The cost (weight) of a constraint C_i induced by an assignment v in which the variables of $S_i = \{X_{i_1}, \ldots, X_{i_k}\}$ take values $b_{i_1}, \ldots, b_{i_k}$ is $f_i(b_{i_1}, \ldots, b_{i_k})$.*

An optimal solution to a WCSP instance is a complete assignment in which the sum of the costs of the constraints not satisfied is minimal.

We call hard *those constraints whose associated cost is infinity,* soft *otherwise.*

Note that in the particular case when all penalties are equal to one, weighted CSP identical to MaxCSP.

Example 4 *Consider $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ where $\mathcal{X} = \{X_1, X_2\}, D(X_1) = D(X_2) = \{1, 2, 3, 4\}$ and $\mathcal{C} = \{(X_1 = X_2, 2), (X_2 < X_1, 3)\}$. A possible solution of this WCSP instance is the assignment $\{(X_1, 2), (X_2, 2)\}$. This assignment satisfies only the constraint C_1 and not the constraint C_2, but 2 is the best possible cost.*

It is very common to talk about extensional WCSPs where such restrictions and weights are defined extensionally (on tuples). In the literature [BMR+99, RL09] there is another definition for (extensional) WCSP (CFN) based on a specific valuation structure $S(k)$.

$S(k)$ is a triple $([0, \ldots, k], \oplus, \geq)$ where:

- $k \in [1, \ldots, \infty]$ is either a strictly positive natural or infinity.
- $[0, \ldots, k]$ is the set of naturals less than or equal to k.
- $\oplus$ is the sum over the valuation structure defined as: $a \oplus b = \min\{k, a + b\}$.
- $\geq$ is the standard order among naturals.

A WCSP is defined by a valuation structure $S(k)$, a set of variables (as for classical CSPs) and a set of constraints. A domain is associated with each variable and a cost function with each constraint. More precisely, for each constraint C and each tuple t that can be built from the domains associated with the variables involved in C, a value in $[0, \ldots, k]$ is assigned to t. When a constraint C assigns the cost k to a tuple t, it means that C forbids t. Otherwise, t is permitted by C with the corresponding cost. The cost of an instantiation of variables is the sum (using operator $\oplus$) over all constraints involving instantiated variables. An instantiation is consistent if its cost is strictly less than k. The goal of the WCSP is to find a full consistent assignment of variables with minimum cost.

Example 5 *The following is an example of an extensional WCSP (4-Queens problem) written in the XCSP 2.1 format (a CSP language described in [RL09]).*

```
<instance>
  <presentation name="4-Queens" maxConstraintArity="2" format="XCSP 2.1"
  type="WCSP" >
    This is the 4-Queens instance represented in extension WCSP.
  </presentation>
  <domains nbDomains="1">
    <domain name="D0" nbValues="4">0..3</domain>
  </domains>
  <variables nbVariables="4">
    <variable name="V0" domain="D0" />
    <variable name="V1" domain="D0" />
    <variable name="V2" domain="D0" />
    <variable name="V3" domain="D0" />
  </variables>
  <relations nbRelations="6">
    <relation name="R0" arity="2" nbTuples="10" semantics="soft"
    defaultCost="0">
```

```
      5:0 0|0 1|1 0|1 1|1 2|2 1|2 2|2 3|3 2|3 3
    </relation>
    <relation name="R1" arity="2" nbTuples="8" semantics="soft"
    defaultCost="0">
      5:0 0|0 2|1 1|1 3|2 0|2 2|3 1|3 3
    </relation>
    <relation name="R2" arity="2" nbTuples="6" semantics="soft"
    defaultCost="0">
      5:0 0|0 3|1 1|2 2|3 0|3 3
    </relation>
    <relation name="R3" arity="1" nbTuples="2" semantics="soft"
    defaultCost="0">
      1:1|3
    </relation>
    <relation name="R4" arity="1" nbTuples="2" semantics="soft"
    defaultCost="0">
      1:1|2
    </relation>
    <relation name="R5" arity="1" nbTuples="2" semantics="soft"
    defaultCost="0">
      1:0|2
    </relation>
  </relations>
  <constraints nbConstraints="10" maximalCost="5">
    <constraint name="C0" arity="2" scope="V0 V1" reference="R0" />
    <constraint name="C1" arity="2" scope="V0 V2" reference="R1" />
    <constraint name="C2" arity="2" scope="V0 V3" reference="R2" />
    <constraint name="C3" arity="2" scope="V1 V2" reference="R0" />
    <constraint name="C4" arity="2" scope="V1 V3" reference="R1" />
    <constraint name="C5" arity="2" scope="V2 V3" reference="R0" />
    <constraint name="C6" arity="1" scope="V0" reference="R3" />
    <constraint name="C7" arity="1" scope="V1" reference="R4" />
    <constraint name="C8" arity="1" scope="V2" reference="R4" />
    <constraint name="C9" arity="1" scope="V3" reference="R5" />
  </constraints>
</instance>
```

In this example we can observe that the restrictions are declared referring to relations. In this case, the relations represent no-goods, since the weight of the listed tuples is > 0 *and the weight of not listed tuples is* 0 *(defaultCost). In this example* k *is* 5 *(maximalCost). Hence, the tuples listed in relations* $R0$*,* $R1$ *and* $R2$ *cannot occur in the solution because they have a weight of* 5*.*

MaxCSP and WCSP are usually solved with branch and bound search [FW92, Wal96]. In recent years there have been many proposals for solving MaxCSP and WCSP, as semirings [BMR97] or maintaining arc consistency [LS04].

Chapter 3

Satisfiability

The Boolean satisfiability problem (SAT) is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to $true$. Equally important is to determine whether no such assignment exists, which would imply that the function expressed by the formula is identically $false$ for all possible variable assignments.

Although SAT was the first known example of an NP-complete problem, in the last decades SAT solvers have spectacularly progressed in performance thanks to better implementation techniques and conceptual enhancements, such as non-chronological backtracking and conflict-driven lemma learning, which in many instances of real problems are able to reduce the size of the search space significantly. Thanks to those advances, nowadays best SAT solvers can tackle problems with hundreds of thousands of variables and millions of clauses. However, no current such methods can efficiently solve all SAT instances.

SAT technology has traditionally been used in the industry for various tasks, software verification and hardware design among them. Some recent examples could be: Windows 7 device drivers are verified using SAT related technology [PdMB10] before being released; also, the Intel® Core™ i7 processor was designed with the aid of SAT technology [KGN+09].

In this chapter we define the SAT problem and present an overview of solving algorithms frequently used in SAT. We also describe variants of the SAT problem such as MaxSAT and Weighted MaxSAT. Finally we describe some techniques to encode CSPs into SAT and its variants.

Sections 3.2, 3.3 and 3.5 are partially based on [Cab11].

3.1 The SAT Problem

Definition 3.1.1 *Let $P = \{x_1, \ldots, x_n\}$ be a finite set of propositional symbols (Boolean variables). Propositional symbols $x_i \in P$ may take values $false$ or $true$. A* literal l_i *is a propositional symbol x_i or its negation $\neg x_i$. The complementary of a literal l, denoted by $\neg l$, is x if $l = \neg x$ and is $\neg x$ if $l = x$.*

A clause *(C) is a disjunction of literals $l_1 \vee \ldots \vee l_n$ and a CNF* formula *ϕ is a conjunction of clauses $C_1 \wedge \cdots \wedge C_m$. A clause is often presented as a set of literals, and a CNF formula as a set of clauses.*

Definition 3.1.2 *The* size of a clause *C, denoted by $|C|$, is the total number of literal occurrences in the clause. A clause with one literal is called unit clause and a clause with two literals is called binary clause. The special case of a clause with zero literals is the empty clause. Empty clauses cannot be satisfied (see below). The size of a CNF formula ϕ, denoted by $|\phi|$, is the sum of the sizes of all its clauses.*

Definition 3.1.3 *A (partial) truth* assignment *M is a set of literals such that $\{l, \neg l\} \not\subseteq M$ for any literal l and $\neg l$ build with the propositional symbols of P. A literal l is $true$ in M if $l \in M$, it is $false$ in M if $\neg l \in M$ and l is $undefined$ in M otherwise. M is total over P if no literal build with the propositionals symbols of P is undefined in M. A clause C is $true$ in M if at least one of its literals is in M (hence, the empty clause cannot be satisfied), it is $false$ in M if all its literals are $false$ in M, and it is undefined in M otherwise.*

Definition 3.1.4 *A formula ϕ is* satisfied *by M (ϕ is $true$ in M), denoted by $M \models \phi$, if all its clauses are $true$ in M. In that case, M is called a* model *of ϕ. If ϕ has no models then it is called* unsatisfiable.

Definition 3.1.5 *The Satisfiability Problem (SAT) for a CNF formula ϕ is the problem of deciding if there exists a truth assignment M that satisfies all the clauses of ϕ.*

Definition 3.1.6 *If ϕ and ϕ' are formulas, we write $\phi \models \phi'$ if ϕ' is $true$ in all models of ϕ. Then we say that ϕ' is* entailed *by ϕ, or is* logical consequence *of ϕ. If $\phi \models \phi'$ and $\phi' \models \phi$, we say that ϕ and ϕ' are* logically equivalent.

Example 6 *Let us consider a CNF formula ϕ having three clauses c_1, c_2 and c_3:*

$$c_1 : x_1 \vee \neg x_2$$
$$c_2 : x_1 \vee x_3$$
$$c_3 : \neg x_1 \vee x_2 \vee x_3$$

Under the partial truth assignment $\{\neg x_1, \neg x_2\}$, clauses c_1 and c_3 are satisfied and clause c_2 is undefined (note that clause c_2 becomes unit, since literal x_1 can be deleted from it). Therefore, the CNF formula ϕ is undefined.

Suppose now that this assignment is completed by adding the literal $\neg x_3$. Then, clause c_2 becomes unsatisfied. Finally, if we consider the assignment $\{\neg x_1, \neg x_2, x_3\}$, all the clauses are satisfied.

We remark that we may use an alternative notion of truth assignment that is focussed on variables instead that on literals. This alternative notion considers assignments as functions from variables to truth values $M : P \rightarrow \{true, false\}$. Then, formulas can be logically evaluated according to them. When the evaluation, according to a given assignment M, of a Boolean formula ϕ is $true$, we say that $M \models \phi$.

3.2 Satisfiability Algorithms

In this section we review some solving techniques which are frequently used in SAT. Namely, we describe some well-known complete algorithms. Complete algorithms perform a search through the space of all possible truth assignments, in a systematic manner, to prove either a given formula is satisfiable (the algorithm finds a satisfying truth assignment) or unsatisfiable (the algorithm completely explores the search space without finding any satisfying truth assignment). By contrast, local search algorithms usually do not completely explore the search space, and a given truth assignment can be considered more than once. We do not discuss the local search algorithms, since this topic is out of the scope of this thesis. Some basic concepts on local search have been described in Section 2.7.

We start by presenting the resolution method, which applies an inference rule that provides a refutation complete inference system. Then, we describe the Davis-Putnam (DP) procedure, the first known effective method for producing resolution refutations. Finally, we present the Davis-Logemann-Loveland (DLL) procedure, implemented in the majority of state-of-the-art complete SAT algorithms, and review the main solving techniques that have been incorporated in DLL in order to devise fast SAT solvers.

3.2.1 The Resolution Method

Resolution is one of the complete methods used to solve SAT. It is based on the resolution rule, which provides a refutation complete inference system [Rob65]. Given two clauses c_1, c_2, called parent clauses, r is a resolvent of c_1 and c_2 if there is one literal $l \in c_1$ such

that $\neg l \in c_2$, and

$$r = (c_1 \setminus \{l\}) \cup (c_2 \setminus \{\neg l\})$$

The resolution step for a CNF formula ϕ, denoted by $Res(\phi)$, is defined as follows:

$$Res(\phi) = \phi \cup \{r \mid r \text{ is a resolvent of two clauses in } \phi\}$$

The resolution procedure consists in computing resolution steps to a formula ϕ until the empty clause is derived or a fix point is reached. If the empty clause is derived, ϕ is unsatisfiable; otherwise, it is satisfiable. Algorithm 7 [Sch89] describes this procedure.

Algorithm 7 Resolution(ϕ): Resolution based SAT algorithm

Input: ϕ : CNF formula
Output: Satisfiability of ϕ

```
repeat
  φ' ← φ
  φ ← Res(φ)
until □ ∈ φ ∨ φ = φ'
if □ ∈ φ then
  return false
else
  return true
end if
```

3.2.2 The Davis-Putnam Procedure

The first effective method for producing resolution refutations was the Davis-Putnam procedure (DP) [DP60]. DP is based on iteratively simplifying the formula until the empty clause is generated (proving unsatifisfiability) or until the formula is empty (proving satisfiability). It consists of three rules:

- **Unit Propagation (UP)**, also referred to as Boolean constraint propagation [ZM88], is the iterated application of the Unit Clause (UC) rule (also referred to as the one-literal rule) until an empty clause is derived or there are no unit clauses left. If a clause is unit, then its single literal must be $true$ under any satisfying assignment. If $\{l\}$ is a unit clause of a CNF formula ϕ, UC consists in deleting all the clauses in ϕ with literal l, and removing all the occurrences of literal $\neg l$ in the remaining clauses.
- **Pure Literal Rule** (also referred to as monotone literal rule). A literal is pure if its complementary literal does not occur in the CNF formula. The satisfiability of

a CNF formula is unaffected by satisfying its pure literals. Therefore, all clauses containing a pure literal can be removed.

- **Resolution** is applied in order to iteratively eliminate each variable from the CNF formula. In order to do so, DP applies a refinement (a restriction) of the resolution method, known as variable elimination. Given the set of clauses C_l containing l and the set of clauses $C_{\neg l}$ containing $\neg l$, the method consists in generating all the non-tautological resolvents using all clauses in C_l and all clauses in $C_{\neg l}$, and then removing all clauses in $C_l \cup C_{\neg l}$. After this step, the CNF formula contains neither l nor $\neg l$.

Algorithm 8 Davis-Putnam(ϕ): DP procedure for SAT

Input: ϕ : CNF formula
Output: Satisfiability of ϕ
  $\phi \leftarrow$ UnitPropagation(ϕ)
  $\phi \leftarrow$ PureLiteralRule(ϕ)
  if $\phi = \emptyset$ **then**
    return $true$
  end if
  if $\square \in \phi$ **then**
    return $false$
  end if
  $l \leftarrow$ literal in $c \in \phi$ having c the minimum length
  $\Re_l \leftarrow$ all possible non-tautological resolvent clauses between all clauses in C_l and all clauses in $C_{\neg l}$
  return Davis-Putnam($\phi \cup \Re_l \setminus (C_l \cup C_{\neg l})$)

The pseudo-code of the DP procedure is given in Algorithm 8. After applying Unit Propagation and the Pure Literal rule, the algorithm selects a variable to be eliminated among the shortest clauses. The procedure stops when the CNF formula is found to be either satisfiable or unsatisfiable. It is declared to be unsatisfiable whenever a conflict (the empty clause) occurs while applying unit propagation. If no conflict occurs, the CNF formula becomes empty and is declared to be satisfiable.

The worst-case memory requirement for DP is exponential. In practice, DP can only handle SAT instances with tens of variables because of this exponential blow-up [Urq87, CS00].

Example 7 *Given the following CNF formula, we demonstrate its satisfiability using the DP algorithm:*

$$(x_1) \wedge (x_1 \vee x_2) \wedge (x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_3 \vee x_5) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_5)$$

We show the steps applied by the DP algorithm. In the first column, the input formula is displayed, with a different clause at each line. The second column represents the result of applying the UC rule with clause x_1. Removed clauses are marked with "$-$" and modified clauses are displayed in bold.

ϕ	x_1
(x_1)	$-$
$(x_1 \vee x_2)$	$-$
$(x_2 \vee x_4)$	$(x_2 \vee x_4)$
$(\neg x_1 \vee x_3 \vee \neg x_4)$	$(\mathbf{x_3} \vee \mathbf{\neg x_4})$
$(x_3 \vee x_5)$	$(x_3 \vee x_5)$
$(\neg x_1 \vee \neg x_3 \vee \neg x_5)$	$(\mathbf{\neg x_3} \vee \mathbf{\neg x_5})$

In a second step, DP applies the pure literal rule. The table below shows the application of the rule to literal x_2, and then to literal $\neg x_4$.

ϕ	x_2	$\neg x_4$
$(x_2 \vee x_4)$	$-$	
$(x_3 \vee \neg x_4)$	$(x_3 \vee \neg x_4)$	$-$
$(x_3 \vee x_5)$	$(x_3 \vee x_5)$	$(x_3 \vee x_5)$
$(\neg x_3 \vee \neg x_5)$	$(\neg x_3 \vee \neg x_5)$	$(\neg x_3 \vee \neg x_5)$

Finally, DP applies resolution. The table below shows the elimination of variable x_3. Observe that a tautological clause appears.

ϕ	x_3
$(x_3 \vee x_5)$	
$(\neg x_3 \vee \neg x_5)$	$(x_5 \vee \neg x_5)$

The tautology is removed by the method and, since the CNF formula becomes empty, the original CNF formula is satisfiable.

3.2.3 The Davis-Logemann-Loveland Procedure

The vast majority of state-of-the-art complete SAT algorithms are built upon the backtrack search algorithm of Davis, Logemann and Loveland (DLL) [DLL62]. DLL replaces the application of resolution in DP by the splitting of the CNF formula into two subproblems. Given a literal l occurring in ϕ, the first subproblem ($\phi_{\neg l}$) is the application of the UC rule over ϕ with $\neg l$, and the second subproblem (ϕ_l) is the application of the UC rule over ϕ

Algorithm 9 Davis-Logemann-Loveland(ϕ): DLL procedure for SAT

Input: ϕ : CNF formula
Output: Satisfiability of ϕ

```
φ ← UnitPropagation(φ)
φ ← PureLiteralRule(φ)
if φ = ∅ then
  return true
end if
if □ ∈ φ then
  return false
end if
l ← literal in c ∈ φ having c the minimum length
return Davis-Logemann-Loveland(φ_l) ∨ Davis-Logemann-Loveland(φ_¬l)
```

with l. Then, ϕ is unsatisfiable if and only if ϕ_l and $\phi_{\neg l}$ are unsatisfiable. This method is shown in Algorithm 9.

The DLL procedure essentially constructs a binary search tree in a depth-first manner. The leaf nodes not containing empty clauses represent complete assignments (i.e., all variables are assigned) while internal nodes represent partial assignments (i.e., some variables are assigned, the rest are free). The DLL procedure explores the search tree and determines that there exists an assignment that satisfies the input formula if the empty formula (that is, the formula contains no clauses) is derived, and that there exists no assignment that satisfies the input formula if all the branches of the search tree contain the empty clause.

DLL incorporates Unit Propagation and the Pure Literal Rule in order to avoid the explicit exponential enumeration of the whole search space. Using a variable selection heuristic, the branching variables are selected to reach a dead-end as early as possible.

Example 8 *In this example we show the search tree for the following CNF formula using the DLL procedure.*

$$\Gamma_0 = \phi \quad : \quad (x_1 \vee x_5) \wedge (x_1 \vee \neg x_6) \wedge (x_1 \vee \neg x_2 \vee x_4) \wedge (x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee x_6)$$

The subproblems associated with each internal node are the following:

$$\Gamma_{10} = \Gamma_{0(\neg x_1)} \quad : \quad (x_5) \wedge (\neg x_6) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_4) \wedge (\neg x_2 \vee \neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_6)$$

$$\Gamma_{11} = \Gamma_{0(x_1)} \quad : \quad (\neg x_2 \vee \neg x_4 \vee \neg x_5) \wedge (x_2 \vee x_4 \vee \neg x_3) \wedge (\neg x_2) \wedge (x_2 \vee x_3 \vee x_6)$$

$$\Gamma_{20} = \Gamma_{10(x_5)} \quad : \quad (\neg x_6) \wedge (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_4) \wedge (x_2 \vee x_4 \vee \neg x_3) \wedge (x_2 \vee x_3 \vee x_6)$$

$$\Gamma_{21} = \Gamma_{11(\neg x_2)} \quad : \quad (x_4 \vee \neg x_3) \wedge (x_3 \vee x_6)$$

$$\Gamma_{30} = \Gamma_{20(\neg x_6)} \quad : \quad (\neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_4) \wedge (x_2 \vee x_4 \vee \neg x_3) \wedge (x_2 \vee x_3)$$

$$\Gamma_{31} = \Gamma_{21(x_4)} \quad : \quad (x_3 \vee x_6)$$

$$\Gamma_{40} = \Gamma_{30(\neg x_2)} \quad : \quad (\neg x_4) \wedge (x_4 \vee \neg x_3) \wedge (x_3)$$

$$\Gamma_{41} = \Gamma_{30(x_2)} \quad : \quad (x_4) \wedge (\neg x_4)$$

$$\Gamma_{50} = \Gamma_{40(x_3)} \quad : \quad (\neg x_4) \wedge (x_4)$$

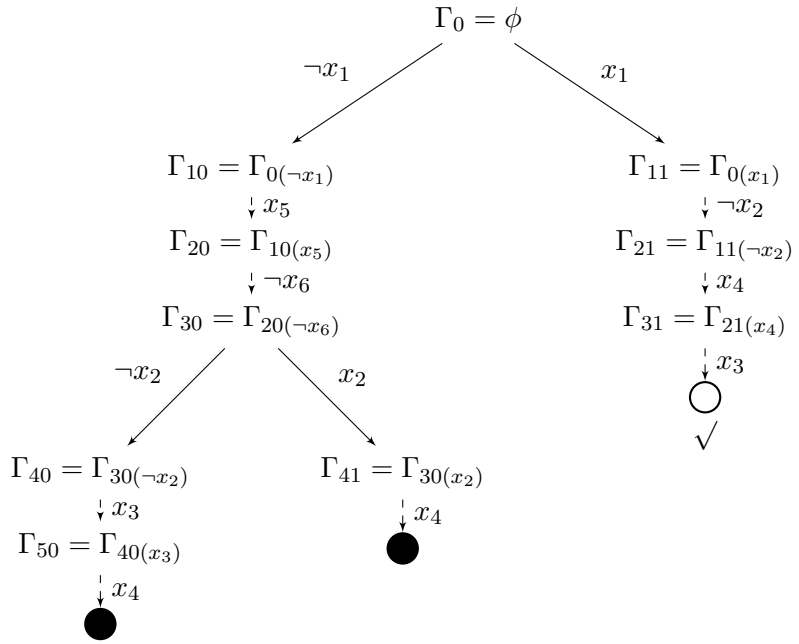


Figure 3.1: Search tree for DLL.

Figure 3.1 shows that the input CNF formula is satisfiable since the empty formula has been reached after assigning all variables.

The authors of [DLL62] identified three advantages of DLL over DP:

1. DP increases the number and length of the clauses rather quickly. DLL never increases the length of clauses.
2. Many redundant clauses may appear after resolution in DP, and seldom after splitting in DLL.
3. DLL often can yield new unit clauses, while DP not often will.

Modern DLL implementations are conflict-driven, that is, the original DLL algorithm is augmented with learning and non-chronological backtracking to facilitate the pruning of the search space. Algorithm 10 shows these so called Conflict-Driven Clause-Learning (CDCL) algorithms.

The auxiliary functions of Algorithm 10 can be defined as follows:

- **UnitPropagation** iteratively applies the unit clause rule. If the empty clause is found, then a conflict indication is returned.
- **PickBranchingVariable** guesses a label, i.e., it selects a variable to assign, and its respective value. The variable selection heuristic is decisive for finding as quick as possible a solution with the DLL procedure [Sil99]. A bad heuristic can lead to explore the whole search space, whereas a good heuristic allows us to cut several regions, and even not to traverse more than a single branch in the best case. The variable is selected after applying unit propagation and the pure literal rule, and is used to split the CNF formula into two subproblems. There are a lot of heuristic methods for select the variable to assign [DPBC93, Pre93, Fre95, JW90].
- **ConflictAnalysisAndLearning** consists of analyzing the most recent conflict, and learning a new clause from the conflict. During the conflict analysis process, the information about the current conflict can be stored by means of redundancy [BS94, BGS99]. These redundant clauses do not change the satisfiability of the original formula, but they help to prune parts of the search space with conflicts that involve variables of the learned conflict. This technique is called clause learning or conflict driven clause learning (see Example 9).
- **NonChronologicalBacktrack** backtracks to the decision level computed by ConflictAnalysisAndLearning. The NonChronologicalBacktrack procedure detects the reason of the conflict and often backtracks to a lower decision level than the previous.

Algorithm 10 Conflict-Driven Clause-Learning algorithm

Input: ϕ : CNF formula
Output: Satisfiability of ϕ

```
dl ← 0 {Decision level}
ν ← ∅ {Current assignment}
UnitPropagation(φ,ν)
if □ ∈ φ then
  return false
end if;
while not AllVariablesAssigned(φ,ν) do
  (x, v) ← PickBranchingVariable(φ, ν)
  dl ← dl + 1
  ν ← ν ∪ {(x, v)}
  UnitPropagation(φ,ν)
  if □ ∈ φ then
    β ← ConflictAnalysisAndLearning(φ,ν)
    if β = 0 then
      return false
    else
      NonChronologicalBacktrack(φ,ν,β)
      dl ← β
    end if
  end if
end while
return true
```

- **AllVariablesAssigned** tests if all variables have been assigned, in which case the algorithm terminates indicating a satisfiable result.

The performance of the DLL procedure critically depends upon the care taken in the implementation. Solvers implementing DLL spend much of their time applying unit propagation [Zha97, LMS05], and this has motivated the definition of several proposals to reduce the cost of applying unit propagation. The most efficient in the state-of-the-art is the unit propagation algorithm based on the called 2-literal watching scheme, first used in Chaff [MMZ$^+$01].

3.2.4 Abstract Davis-Putnam-Logemann-Loveland

The Davis-Putnam procedure was originally presented as a two-phase proof-procedure for first-order logic. The unsatisfiability of a formula was to be proved by first generat-

ing a suitable set of ground instances which then, in the second phase, were shown to be propositionally unsatisfiable. Subsequent improvements, such as the Davis-Logemann-Loveland procedure, mostly focused on the propositional phase. What most authors nowadays call the Davis-Putnam-Logemann-Loveland (DPLL) procedure is a satisfiability procedure for propositional logic based on this propositional phase.

A DPLL procedure can be modelled by a transition relation over states [NOT06]. A state is either $FailState$ or a pair $M \parallel \phi$, where ϕ is a finite set of clauses and M is a partial assignment (in the form of a sequence of literals). Some literals l in M will be annotated as being decision literals; these are the ones added to M by the $Decide$ rule, and are written l^d. The transition relation is defined by means of rules.

The classical DPLL transition system consists of the following five rules:

$UnitPropagate$:

$$M \parallel \phi, C \vee l \implies Ml \parallel \phi, C \vee l \quad \text{if} \quad \begin{cases} M \models \neg C \text{ and} \\ l \text{ is undefined in } M. \end{cases}$$

$PureLiteral$:

$$M \parallel \phi \implies Ml \parallel \phi \quad \text{if} \quad \begin{cases} l \text{ occurs in some clause of } \phi, \\ \neg l \text{ occurs in no clause of } \phi \text{ and} \\ l \text{ is undefined in } M. \end{cases}$$

$Dedide$:

$$M \parallel \phi \implies Ml^d \parallel \phi \quad \text{if} \quad \begin{cases} l \text{ or } \neg l \text{ occurs in some clause of } \phi \text{ and} \\ l \text{ is undefined in } M. \end{cases}$$

$Fail$:

$$M \parallel \phi, C \implies FailState \quad \text{if} \quad \begin{cases} M \models \neg C \text{ and} \\ M \text{ contains no decision literals.} \end{cases}$$

$Backtrack$:

$$Ml^dN \parallel \phi, C \implies M\neg l \parallel \phi, C \quad \text{if} \quad \begin{cases} Ml^dN \models \neg C \text{ and} \\ N \text{ contains no decision literals.} \end{cases}$$

The majority of modern DPLL algoritms only use the $PureLiteral$ rule as a preprocessing step, replace the $Backtrack$ rule by the $Backjump$ rule and add tree new rules: the $Learn$ rule which implements the conflict-driven clause-learning, the $Forget$ rule for forgetting learned clauses (usually for reasons of space), and the $Restart$ rule that restarts the algorithm but remembering what has been learned. Hopefully, these newly learned lemmas will lead the heuristics for $Decide$ to behave differently.

Backjump :

$$Ml^d N \parallel \phi, C \implies Ml' \parallel \phi, C \quad \text{if} \quad \begin{cases} Ml^d N \models \neg C \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \phi, C \models C' \vee l' \text{ and } M \models \neg C', \\ l' \text{ is undefined in } M, \text{ and} \\ l' \text{ or } \neg l' \text{ occurs in } \phi \text{ or in } Ml^d N. \end{cases}$$

Learn :

$$M \parallel \phi \implies M \parallel \phi, C \quad \text{if} \quad \begin{cases} \text{all atoms of } C \text{ occur in } \phi \text{ or in } M \text{ and} \\ \phi \models C. \end{cases}$$

Forget :

$$M \parallel \phi, C \implies M \parallel \phi \quad \text{if} \quad \phi \models C.$$

Restart :

$$M \parallel \phi \implies \emptyset \parallel \phi.$$

Example 9 *Aplication of the DPLL rules on the formula* $(\neg x_1 \vee x_2) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_5 \vee \neg x_6) \wedge (x_6 \vee \neg x_5 \vee \neg x_2)$.

$\emptyset$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*Decide*
x_1^d	$\parallel$	$\underline{\neg x_1 \vee x_2}, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*UnitPropagate*
$x_1^d\ x_2$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*Decide*
$x_1^d\ x_2\ x_3^d$	$\parallel$	$\neg x_1 \vee x_2, \underline{\neg x_3 \vee x_4}, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*UnitPropagate*
$x_1^d\ x_2\ x_3^d\ x_4$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*Decide*
$x_1^d\ x_2\ x_3^d\ x_4\ x_5^d$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \underline{\neg x_5 \vee \neg x_6}, x_6 \vee \neg x_5 \vee \neg x_2$	$\Rightarrow$	*UnitPropagate*
$x_1^d\ x_2\ x_3^d\ x_4\ x_5^d\ \neg x_6$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, \underline{x_6 \vee \neg x_5 \vee \neg x_2}$	$\Rightarrow$	*Backjump & Learn*
$x_1^d\ x_2\ \neg x_5$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2, \neg x_2 \vee \neg x_5$	$\Rightarrow$	*Decide*
$x_1^d\ x_2\ \neg x_5\ x_3^d$	$\parallel$	$\neg x_1 \vee x_2, \underline{\neg x_3 \vee x_4}, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2, \neg x_2 \vee \neg x_5$	$\Rightarrow$	*UnitPropagate*
$x_1^d\ x_2\ \neg x_5\ x_3^d\ x_4$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2, \neg x_2 \vee \neg x_5$	$\Rightarrow$	*Decide*
$x_1^d\ x_2\ \neg x_5\ x_3^d\ x_4\ \neg x_6^d$	$\parallel$	$\neg x_1 \vee x_2, \neg x_3 \vee x_4, \neg x_5 \vee \neg x_6, x_6 \vee \neg x_5 \vee \neg x_2, \neg x_2 \vee \neg x_5$	$\Rightarrow$	*Solution*

We underline the clause causing the $Backjump$ *or* $UnitPropagation$.

Note that, in the $Backjump$ step, $x_6 \vee \neg x_5 \vee \neg x_2$ is the conflicting clause C, since all its literals are false under the partial assignment $x_1\ x_2\ x_3\ x_4\ x_5\ \neg x_6$. Then, roughly, since neither x_3 nor x_4 occur in the conflicting clause, the $Backjump$ rule allows us to $Backjump$ to the state $x_1\ x_2\ \neg x_5 \parallel \ldots$ Observe that, in this case, $M = x_1\ x_2$, $l^d = x_3$, $N = x_4\ x_5\ \neg x_6$, $C = x_6 \vee \neg x_5 \vee \neg x_2$, and $l' = \neg x_5$. We can take $C' \vee l' = \neg x_2 \vee \neg x_5$. Note that this clause can be obtained by resolution between $\neg x_5 \vee \neg x_6$ (the clause that has propagated $\neg x_6$) and the conflicting clause $x_6 \vee \neg x_5 \vee \neg x_2$:

$$\frac{\neg x_5 \vee \neg x_6 \qquad\qquad x_6 \vee \neg x_5 \vee \neg x_2}{\neg x_2 \vee \neg x_5}$$

Note also that $C' \vee l'$ becomes unit under assignment M (and hence l' can be propagated). Moreover, this clause can be learned according to the $Learn$ rule, since resolution is sound and hence it is a logical consequence of the original formula. Learning this clause will prevent similar conflicts in the future.

On the other hand, the indiscriminate application of the $restart$ rule could lead to incompleteness. To avoid incompleteness, it suffices to perform restarts at increasing periods of time (typically counting up to a certain number of conflicts). But many other types of restart policies have been studied, such as arithmetic o geometric series over the number of conflicts, or especial series such as the Luby series or Inner-Outer Geometric series [Hua07, RS08].

Note that any of these rules can be executed whenever its precondition is met. The particular order will be determined by the specific solver using these rules.

3.3 MaxSAT and Weighted MaxSAT

In this section we describe optimization variants of the SAT problem such as MaxSAT, Partial MaxSAT, Weighted MaxSAT and Partial Weighted MaxSAT. These variants can be used to obtain the best possible truth assignment for unsatisfiable formulas.

3.3.1 MaxSAT and Partial MaxSAT

Given a CNF formula, the maximum satisfiability (MaxSAT) problem is defined as the problem of finding a solution that maximizes the number of satisfied clauses.

In the Partial MaxSAT problem there is a set of clauses that can be violated (soft clauses) and a set of clauses that must be satisfied (hard clauses). The objective of the problem is also the maximization of the number of satisfied clauses, or equivalently, the minimization of the number of falsified clauses (cost), but ensuring the satisfaction of all the hard clauses.

In SAT, a CNF formula is considered to be a set of clauses, while in (partial) MaxSAT, a CNF formula is considered to be a multiset of clauses, because repeated clauses cannot be collapsed into a unique clause. For instance, in $\{x_1, \neg x_1, \neg x_1, x_1 \vee x_2, \neg x_2\}$, where a clause is repeated, there is a minimum of two unsatisfied clauses. If we consider the set $\{x_1, \neg x_1, x_1 \vee x_2, \neg x_2\}$, where repeated clauses are collapsed, then there is a minimum of one unsatisfied clause.

Example 10 *Let us consider a Partial MaxSAT instance ϕ having the following clauses:*

$$
\begin{aligned}
&c_1 : [x_1 \vee x_2] \\
&c_2 : [\neg x_1] \\
&c_3 : [\neg x_1 \vee \neg x_2] \\
&c_4 : (\neg x_2 \vee x_3) \\
&c_5 : (x_1 \vee \neg x_2) \\
&c_6 : (\neg x_3) \\
&c_7 : (x_1 \vee \neg x_2 \vee \neg x_3)
\end{aligned}
$$

Hard clauses are represented between square brackets, and soft clauses are represented between round brackets.

An optimal solution for ϕ is the assignment $\{\neg x_1, x_2, \neg x_3\}$, which satisfies all the hard clauses and maximizes the number of satisfied soft clauses. The number of falsified soft clauses (cost) is 2, and the satisfied soft clauses are c_6 and c_7.

3.3.2 Weighted MaxSAT and Weighted Partial MaxSAT

A Weighted CNF clause is a pair (C, w), where C is a clause and w is its weight. The weight w can be a natural number or infinity and its meaning is the penalty (cost) for falsifying the clause C. A clause is hard if its corresponding weight is infinity, and soft otherwise. When there are no hard clauses we speak of Weighted MaxSAT, and we speak of Weighted Partial MaxSAT otherwise. A Weighted CNF formula is a multiset of weighted clauses.

The Weighted MaxSAT problem consists of finding a truth assignment such that the sum of the weights of the satisfied clauses is maximized, or equivalently, the total weight of the falsified clauses is minimized.

Example 11 *Let us consider a Weighted MaxSAT instance having the following clauses:*

$$
\begin{aligned}
&c_1 : (x_1 \vee x_2, 4) \\
&c_2 : (\neg x_1, 3) \\
&c_3 : (\neg x_1 \vee \neg x_2, 5) \\
&c_4 : (\neg x_2 \vee x_3, 3) \\
&c_5 : (x_1 \vee \neg x_2, 2) \\
&c_6 : (\neg x_3, 1)
\end{aligned}
$$

An optimal solution for this formula is the assignment $\{\neg x_1, x_2, x_3\}$*. The sum of weights of satisfied clauses is* 15*, and the sum of weights of falsified clauses is* 3*. This assignment falsifies the clauses* c_5 *and* c_6*.*

We remark that the MaxSAT problem can also be defined as Weighted MaxSAT restricted to formulas whose clauses have weight 1, and as Partial MaxSAT in the case that all the clauses are declared to be soft. The Partial MaxSAT problem is Weighted Partial MaxSAT when the weights of the soft clauses are equal. Note also that the SAT problem is equivalent to Partial MaxSAT when there are no soft clauses.

The notion of equivalence with respect to instances has some subtlety that we want also to remark. In SAT, two formulas are equivalent if they are satisfied by the same set of assignments. In MaxSAT, two formulas are equivalent if both have the same number of unsatisfied clauses for every assignment. In Weighted MaxSAT, two formulas are equivalent if the sum of the weights of unsatisfied clauses coincides for every assignment.

3.4 MaxSAT and Weighted MaxSAT Algorithms

In recent years several methods have been described to solve MaxSAT and Weighted MaxSAT.

A first approach could consist in reifying the soft clauses into Boolean variables. These Boolean variables can be considered as pseudo-Boolean, i.e., $\{0, 1\}$ integer variables, corresponding 0 to $false$ and 1 to $true$. The objective function in the case of MaxSAT is the sum of these pseudo-Boolean variables and in the case of Weighted MaxSAT the sum of products of the pseudo-Boolean variable by the corresponding clause weight. In both cases we must minimize the objective function.

Another possibility widely used in recent times (winning in several categories of the last SAT competitions) is to achieve the optimal solution using unsatisfiability cores (UNSAT cores). We look in some more detail to this kind of algorithms since the use of these methods for SMT is one of the thesis contributions.

3.4.1 UNSAT Core Based Algorithms

Definition 3.4.1 *An* unsatisfiable core *is an unsatisfiable subset of clauses* ϕ_c *of the original CNF* ϕ*.*

In the last few years, several algorithms for computing small [ZM03], minimal [DHN06] or minimum [ZLS06] unsatisfiable cores of propositional formulas have been proposed.

Currently there are a large number of UNSAT core based algorithms and numerous variants of each. We describe some of them:

- **PM1** was proposed by [FM06] for the Partial MaxSAT problem. This algorithm consists in iteratively calling a SAT solver on a working formula ϕ (see Algorithm 11). The SAT solver will say whether the formula is satisfiable or not, and in case the formula is unsatisfiable, it will return us an unsatisfiable core (ϕ_c). At this point the algorithm will create new variables (BV), called blocking variables, one for each clause of the returned core. Then, the new working formula ϕ will consist of the previous ϕ where the each new variable has been added to the corresponding clause of the core. Moreover, a cardinality constraint saying that exactly one of the new variables should be $true$ $\{C|C \in CNF(\sum_{b \in BV} ite(b; 1; 0) = 1)\}$ is also added. Finally, it is increased by one the counter of falsified clauses ($cost$). This procedure is applied until the SAT solver returns satisfiable.

Algorithm 11 PM1 Algorithm

Input: ϕ : CNF formula
Output: Cost of ϕ

```
cost ← 0
while true do
  (st, φ_c) ← SAT_ALGORITHM(φ)
  if st = SAT then
    return cost
  else
    BV ← ∅
    for all C ∈ φ_c do
      if isSoft(C) then
        b ← New_variable()
        φ ← φ \ {C} ∪ {C ∨ b}
        BV ← BV ∪ {b}
      end if
    end for
    if BV = ∅ then
      return UNSAT
    end if
    φ ← φ ∪ {C|C ∈ CNF(∑_{b∈BV} ite(b; 1; 0) = 1)}
    cost ← cost + 1
  end if
end while
```

- **WPM1** was proposed by [ABL09] for the Weighted MaxSAT. This is the natural extension of PM1 for Weighted MaxSAT (see Algorithm 12).

Algorithm 12 WPM1 Algorithm

Input: $\phi = \{(C_1, w_1), \ldots, (C_n, w_n)\}$: CNF formula
Output: Cost of ϕ

```
cost ← 0
while true do
  (st, φ_c) ← SAT_ALGORITHM({C_i | (C_i, w_i) ∈ φ})
  if st = SAT then
    return cost
  else
    BV ← ∅
    w_min ← min{w_i | C_i ∈ φ_c ∧ isSoft(C_i)}
    for all C_i ∈ φ_c do
      if isSoft(C_i) then
        b ← New_variable()
        φ ← φ \ {(C_i, w_i)} ∪ {(C_i, w_i − w_min)} ∪ {(C_i ∨ b, w_min)}
        {when w_i − w_min = 0 the clause (C_i, w_i − w_min) is not added}
        BV ← BV ∪ {b}
      end if
    end for
    if BV = ∅ then
      return UNSAT
    end if
    φ ← φ ∪ {(C, ∞)|C ∈ CNF(∑_{b∈BV} ite(b; 1; 0) = 1)}
    cost ← cost + w_min
  end if
end while
```

Example 12 *Let us consider a Weighted MaxSAT instance ϕ having the following clauses: $\{(x,1),(y,2),(z,3),(\neg x \vee \neg y,\infty),(x \vee \neg z,\infty),(y \vee \neg z,\infty)\}$. We show the application of the WPM1 algorithm on ϕ in Figure 3.2.*

- **PM2** was proposed by [ABL09] where a single blocking variable is added to each soft clause.

1st. iter.		2nd. iter.		3rd. iter.		4th. iter.	
Clauses	W.	Clauses	W.	Clauses	W.	Clauses	W.
$\underline{x}$	1						
y	2	$\underline{y}$	2				
$\underline{z}$	3	$\underline{z}$	2				
		$x \vee b_1$	1	$\underline{x \vee b_1}$	1		
		$z \vee b_2$	1	$\underline{z \vee b_2}$	1	$z \vee b_2$	1
				$\underline{y \vee b_3}$	2	$y \vee b_3$	1
				$\underline{z \vee b_4}$	2	$z \vee b_4$	1
						$x \vee b_1 \vee b_5$	1
						$y \vee b_3 \vee b_6$	1
						$z \vee b_4 \vee b_7$	1
$\underline{\neg x \vee \neg y}$	∞	$\underline{\neg x \vee \neg y}$	∞	$\underline{\neg x \vee \neg y}$	∞	$\neg x \vee \neg y$	∞
$x \vee \neg z$	∞	$\underline{x \vee \neg z}$	∞	$\underline{x \vee \neg z}$	∞	$x \vee \neg z$	∞
$\underline{y \vee \neg z}$	∞	$y \vee \neg z$	∞	$\underline{y \vee \neg z}$	∞	$y \vee \neg z$	∞
		$CNF(b_1 + b_2 = 1)$	∞	$\underline{CNF(b_1 + b_2 = 1)}$	∞	$CNF(b_1 + b_2 = 1)$	∞
			∞	$\underline{CNF(b_3 + b_4 = 1)}$	∞	$CNF(b_3 + b_4 = 1)$	∞
						$CNF(b_5 + b_6 + b_7 = 1)$	∞
$cost = 0$		$cost = 1$		$cost = 3$		$cost = 4$	

Figure 3.2: Application of the WPM1 algorithm
Application of the WPM1 algorithm on the Weighted CNF $\{(x, 1), (y, 2), (z, 3), (\neg x \vee \neg y, \infty), (x \vee \neg z, \infty), (y \vee \neg z, \infty)\}$. UNSAT cores of each stage are marked in bold and underlined.

3.5 Encoding CSPs into SAT

Recall that a CSP is defined as a triple $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$, where $\mathcal{X} = \{X_1, \ldots, X_n\}$ is a set of variables, $\mathcal{D} = \{D(X_1), \ldots, D(X_n)\}$ is a set of domains containing the values the variables may take, and $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of constraints. Thus, to translate a CSP instance into a SAT instance we must must first choose an encoding for the variables and their values. Restrictions must be translated using these encoding in such a way that they retain their meaning.

In this section we describe several well-known CSP to SAT encodings. For the sake of simplicity, we assume that the domain of all variables is $\{1 \ldots n\}$.

3.5.1 Variable Encodings

CSP instances contain variables whose domain size can be greater than 2 while the Boolean variables can only be $false$ or $true$. Given a CSP variable X, the most frequent way of encoding X into SAT is associating a Boolean variable x_i with each value $i \in D(X)$ in such a way that x_i is $true$ iff $X = i$. These variables are called normal or standard

variables. Moreover, since CSP assignments assign exactly one value of the domain to each variable, we have to encode that exactly one of the Boolean variables $\{x_1, \ldots, x_n\}$ takes the value $true$, and the other variables take the value $false$. This is the goal of the **Exactly-One** constraint, which allows to maintain a one-to-one mapping between CSP models and SAT models.

The Exactly-One constraint is commonly expressed as the conjunction of the ALO (At-Least-One) constraint, and the AMO (At-Most-One) constraint. The ALO constraint states that at least one of the variables is $true$, and the AMO constraint states that at most one of the variables is $true$. There are different encodings of the ALO and AMO constraints. The standard encoding of the ALO constraint is formed by the following n-ary clause:

$$x_1 \vee \ldots \vee x_n$$

And for the AMO constraint, the standard one is the pair-wise encoding, also called naive encoding in the literature. The encoding is formed by the following $n * (n-1)/2$ binary clauses:

$$\bigwedge_{i=1}^{n-1} \bigwedge_{j=i+1}^{n} \neg x_i \vee \neg x_j$$

Other encodings for the AMO constraint are the following:

- **The Binary AMO**, introduced in [FP01], introduces new variables $b_1, \ldots, b_{\lceil log_2 n \rceil}$. It then associates with each x_i a unique bit string $s_i \in \{1, 0\}^{\lceil log_2 n \rceil}$ where the string s_i is the binary representation of i. The binary AMO is:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=i}^{\lceil log_2 n \rceil} \neg x_i \vee \psi(i, j)$$

  where $\psi(i, j)$ is a clause with b_j if the j_{th} bit of s_i is 1 and with $\neg b_j$ otherwise.

  This encoding introduces $\lceil log_2 n \rceil$ extra variables and uses $n * \lceil log_2 n \rceil$ clauses.

- **The Sequential counter AMO**, introduced in [Sin05], is based on a sequential counter circuit, that consists in sequentially counting the number of x_is that are $true$. The Sequential counter AMO is:

$$(\neg x_1 \vee s_1)$$

$$\bigwedge_{i=2}^{i<n} ((\neg x_i \vee s_i) \wedge (\neg s_{i-1} \vee s_i) \wedge (\neg x_i \vee \neg s_{i-1}))$$

$$(\neg x_n \vee \neg s_{n-1})$$

where s_i, $1 \leq i \leq n-1$, are auxiliary variables.

This encoding requires $3n-4$ binary clauses and $n-1$ auxiliary variables.

3.5.2 Constraint Encodings

There are also several possibilities for encoding the CSP constraints. The more relevant ones are:

- **The Direct Encoding** is probably the most popular encoding from CSP into SAT. In the standard direct encoding for each binary constraint with scope $\{X, Y\}$ there is a binary clause for every nogood. Such clauses are called conflict clauses.

  Example 13 *The conflict clauses in the standard direct encoding for the CSP defined by* $\langle\{X, Y\}, \{D(X) = \{1, 2, 3\}, D(Y) = \{1, 2, 3\}\}, \{X \leq Y\}\rangle$ *are the following:*

  $$(\neg x_2 \vee \neg y_1) \quad (\neg x_3 \vee \neg y_1) \quad (\neg x_3 \vee \neg y_2)$$

- **The Support Encoding** encodes into clauses the support for each possible value of a variable across a constraint. The support for a value i of a CSP variable X across a binary constraint with scope $\{X, Y\}$ is the set of values of the variable Y which allow $X = i$.

  Example 14 *The support clauses for the CSP of Example 13 are:*

  $$(\neg x_2 \vee y_2 \vee y_3) \quad (\neg x_3 \vee y_3) \quad (\neg y_1 \vee x_1) \quad (\neg y_2 \vee x_1 \vee x_2)$$

  The support clause for $X = 1$ *is missing because it is subsumed by* $(y_1 \vee y_2 \vee y_3)$*, and the support clause for* $Y = 3$ *is missing because it is subsumed by* $(x_1 \vee x_2 \vee x_3)$*.*

- **The Multivalued Encoding** is a variant of the direct encoding in which the AMO clauses can be omitted. In this case, each CSP variable can take more than one value simultaneously. As pointed out in [BHvMW09], AMO clauses can be omitted in the direct encoding from CSP into SAT in such a way that the CSP is satisfiable if and only if the resulting SAT encoding is satisfiable.

3.5.3 Other Encodings

There are other encodings in which there is no direct relationship between each value $i \in D(X)$ and a Boolean variable x_i, and encodings where in addition to the Boolean variables x_i some other Boolean variables are added. Some of these encodings are the following:

- **Log Encoding** [IM94]. Given a CSP variable X with domain $D(X)$, the log encoding requires $\lceil log_2|D(X)|\rceil$ variables to encode each possible value for X. In this encoding to facilitate the formulation we assume that the domains range from 0 to n.

  In the log encoding, the assignment $X = i$ is encoded by the clause $l^i_1 \vee \ldots \vee l^i_{\lceil log_2|D(X)|\rceil}$, where $l^i_1, \ldots, l^i_{\lceil log_2|D(X)|\rceil}$ are the literals associated with the binary representation of the value i. The log encoding is formed by the conflict clause set, which encodes the no-goods of the constraints, and the prohibited-value clauses if there are domains whose size is not a power of 2. Moreover, in contrast to the direct and support encoding, neither ALO nor AMO clauses are required.

  Example 15 *The log encoding for the CSP* $\langle\{X, Y\}, \{D(X) = \{0, 1, 2\}, \{D(Y) = \{1, 2, 3\}, \{X \neq Y\}\rangle$ *goes as follows:*

value domain (i)	b_1	b_0	*literals associated with* $X = i$	
$X = 0$	0	0	$\neg x_1$	$\neg x_0$
$X = 1$	0	1	$\neg x_1$	x_0
$X = 2$	1	0	x_1	$\neg x_0$
$X = 3$	1	1	x_1	x_0

conflict values	*conflict clauses*
$X = 0, Y = 0$	$x_0 \vee x_1 \vee y_0 \vee y_1$
$X = 1, Y = 1$	$\neg x_0 \vee x_1 \vee \neg y_0 \vee y_1$
$X = 2, Y = 2$	$x_0 \vee \neg x_1 \vee y_0 \neg \vee y_1$

prohibited values	*prohibited-value clauses*
$\neg[X = 3]$	$\neg x_0 \vee x_1$
$\neg[Y = 3]$	$\neg y_0 \vee y_1$

In the first table we can observe the sets of literals associated with the possible assignments of X *(it is similar for the possible assignments of the CSP variable Y).*

In the second table there are the conflict clauses and in the last table the prohibited-value clauses.

- **The Order Encoding** [CB94, BB03]. In this encoding, for every CSP variable X and every value $i \in D(X)$ except for the lower bound (the smallest value of the domain, in our case 1) we associate a Boolean variable $x_i^{\geq}$, in such a way that $x_i^{\geq}$ is $true$ iff $X \geq i$. To encode the AMO and ALO constraints, it is sufficient that these Boolean variables constitute a monotonic non-increasing sequence. To achieve this property we must add the following clauses:

$$\bigwedge_{i=n}^{i>2} \neg x_i^{\geq} \vee x_{i-1}^{\geq}$$

  The encoding of the constraints in this case is quite simple. For example to state $X \neq 2$ we only need to encode the equality $x_2^{\geq} = x_3^{\geq}$ and for the constraint $X < 4$ we only need the clause $\neg x_4^{\geq}$.

- **The Regular Encoding** [AM04, BHM01]. In this encoding, for every CSP variable X and every value $i \in D(X)$, we associate a variable x_i, and a Boolean variable $x_i^{\leq}$ called regular variable (for i equal to the upper bound of $D(X)$ it is not necessary), in such a way that $x_i^{\leq}$ is $true$ iff $X \leq i$. Regular encodings contain both standard and regular variables. To achieve ALO and AMO constraints, and to achieve channelling between standard variables and regular variables we must add the following clauses:

$$\bigwedge_{i=1}^{i<n-1} \neg x_i^{\leq} \vee x_{i+1}^{\leq}$$

$$\bigwedge_{i=1}^{i<n} \neg x_i \vee x_i^{\leq}$$

$$\bigwedge_{i=2}^{i\leq n} \neg x_i \vee \neg x_{i-1}^{\leq}$$

$$x_1 \vee \neg x_1^{\leq}$$

$$\bigwedge_{i=2}^{i<n} x_i \vee \neg x_i^{\leq} \vee x_{i-1}^{\leq}$$

$$x_n \vee \neg x_{n-1}^{\leq}$$

  The first clause encodes the transitivity of the regular variables $x_d^{\leq} \rightarrow x_{d+1}^{\leq}$ and the other five encode the channeling between the standard and the regular variables

$x_d \leftrightarrow x_d^{\leq} \wedge \neg x_{d-1}^{\leq}$. In this encoding neither ALO nor AMO clauses are required because the clauses force that each variable has one and only one value of the domain.

Note that we could use $x_i^{\geq}$ instead of $x_i^{\leq}$ as in the case of the order encoding with the corresponding changes in the clauses.

The encoding of the constraints in this case is quite simple, since they can use both standard variables and regular variables.

Chapter 4

Satisfiability Modulo Theories

An SMT formula is a generalization of a Boolean formula in which some propositional variables have been replaced by predicates with predefined interpretations from background theories such as, e.g., linear integer arithmetic. For example, a formula can contain clauses like $p \vee q \vee (x+2 \leq y) \vee (x > y+z)$, where p and q are Boolean variables and x, y and z are integer variables. Predicates over non-Boolean variables, such as linear integer inequalities, are evaluated according to the rules of a background theory [NOT06, Seb07]. Examples of theories include linear real or integer arithmetic, arrays, bit vectors, uninterpreted functions, etc., or combinations of them.

Leveraging the advances made in SAT solvers in the last decade, SMT solvers have proved to be competitive with classical decision methods in many areas. Most modern SMT solvers integrate a SAT solver with specialized solvers for a set of literals belonging to each theory.

In this chapter we define the SMT problem and present the lazy SMT approach and some of the most used background theories. We also present variants of the SMT problem such as MaxSMT and Weighted SMT.[1] Finally, we describe the Lazy Clause Generation system for solving CSPs, which uses very similar techniques to the ones of SMT.

4.1 Preliminaries

Definition 4.1.1 *A* theory *is a set of first-order formulas closed under logical consequence. A theory T is said to be* decidable *if there is an effective method for determining whether arbitrary formulas are included in T.*

Definition 4.1.2 *A formula φ is T-*satisfiable or T-consistent *if $T \cup \{\varphi\}$ is satisfiable in*

[1] Also called Weighted MaxSMT in the literature.

the first-order sense. Otherwise, it is called T-unsatisfiable or T-inconsistent.

Definition 4.1.3 *A (partial) truth assignment* M *(see Definition 3.1.3) can be seen either as a set or as a conjunction of literals, and hence as a formula. If* M *is a* T*-consistent partial truth assignment and* φ *is a formula such that* $M \models \varphi$*, i.e.,* M *is a (propositional) model of* φ*, then we say that* M *is a* T-model *of* φ*. Moreover, we write* $\varphi \models_T \varphi'$ *as an abbreviation for* $T \cup \{\varphi\} \models \varphi'$*.*

Definition 4.1.4 *The* SMT problem for a theory T *is the problem of determining, given a formula* φ*, whether* φ *is* T*-satisfiable, or, equivalently, whether* φ *has a* T*-model.*

As usually done in SMT, here we only consider the SMT problem for ground (and hence quantifier-free) CNF formulas. Such formulas may contain constants that are free in T, which, as far as satisfiability is concerned, can equivalently be seen as existentially quantified variables.

Moreover, we will only consider theories T for which the T-satisfiability of conjunctions of such ground literals is decidable. A decision procedure for this problem is called a T-solver.

4.2 The Eager and Lazy SMT Approaches

In the so-called eager approach to SMT, the input formula is translated in a single satisfiability preserving step into a propositional CNF formula which is then checked by a SAT solver for satisfiability. Sophisticated ad-hoc translations have been developed for several theories, but still, on many practical problems the translation process or the SAT solver run out of time or memory [dMR04].

Currently most successful SMT solvers are based on the integration of a SAT solver and a T-solver, that is, a decision procedure for the given theory T. In this so-called lazy approach, while the SAT solver is in charge of the Boolean component of reasoning, the T-solver deals with sets of atomic constraints (literals) in T. The basic idea is to let the T-solver analyze the partial truth assignment that the SAT solver is building, and warn about conflicts with theory T (T-inconsistency). This way, we are hopefully getting the best of both worlds: in particular, the efficiency of the SAT solver for the Boolean reasoning and the efficiency of special-purpose algorithms inside the T-solver for the theory reasoning. This approach is usually orders of magnitude faster than the eager approach. It is called lazy because the theory information is only used from time to time, when checking the consistency of the truth assignment with theory T.

Algorithm 13 shows a simplified version of an enumeration-based T-satisfiability procedure (from [BCF$^+$06]) where T-consistency is only checked for total Boolean assignments. We refer the reader to [Seb07] for a survey on the lazy SMT approach.

Algorithm 13 Bool+T

Input: φ : SMT formula
Output: Satisfiability of φ

```
A^p ← T2B(Atoms(φ));
φ^p ← T2B(φ);
while Bool-satisfiable(φ^p) do
  μ^p ← pick_total_assignment(A^p, φ^p);
  (ρ, π) ← T-satisfiable(B2T(μ^p));
  if ρ = sat then
    return sat;
  else
    φ^p ← φ^p ∧ ¬T2B(π);
  end if;
end while
return unsat;
```

The algorithm enumerates the Boolean models of the propositional abstraction of the SMT formula φ and checks for their satisfiability in the theory T.

- The function $Atoms$ takes a quantifier-free SMT formula φ and returns the set of atoms which occur in φ.
- The function $T2B$ maps propositional variables to themselves, and ground atoms into fresh propositional variables, and is homomorphic with respect to Boolean operators and set inclusion. φ^p is initialized to be the propositional abstraction of φ using $T2B$.
- The function $B2T$ is the inverse of $T2B$.
- μ^p denotes a propositional assignment as a set (conjunction) of propositional literals.
- The function $pick\_total\_assignment$ returns a total assignment to the propositional variables in φ^p. In particular, it assigns a truth value to all variables in A^p.
- The function T-$satisfiable$ checks if a set of conjuncts μ is T-$satisfiable$, i.e., if there is a model for $T \cup \mu$, returning (sat,$\emptyset$) in the positive case and (unsat,π) otherwise, being $\pi \subseteq \mu$ a T-$unsatisfiable$ set (the theory conflict set). Note that

the negation of the propositional abstraction of π is added to φ^p in case of `unsat` (learning).

We illustrate Algorithm 13 with Example 16.

Example 16 *Consider the following SMT formula, expressed as a set of clauses, where T is assumed to be the theory of linear integer arithmetic:*

$$\begin{aligned}\varphi = \{&\neg(x > 0) \vee a \vee b,\\ &\neg a \vee \neg b,\\ &\neg(x + 1 < 0) \vee a,\\ &\neg b \vee \neg(y = 1)\}\end{aligned}$$

Then $\{x > 0, a, b, x + 1 < 0, y = 1\}$ is its set of atoms and

$$A^p = \{p_{(x>0)}, a, b, p_{(x+1<0)}, p_{(y=1)}\}$$

is the Booleanization of this set, where $p_{(x>0)}, p_{(x+1<0)}$ and $p_{(y=1)}$ are three fresh propositional variables corresponding to the arithmetic atoms $x > 0, x + 1 < 0$ and $y = 1$, respectively. The propositional abstraction of φ is then the following Boolean formula:

$$\begin{aligned}\varphi^p = \{&\neg p_{(x>0)} \vee a \vee b,\\ &\neg a \vee \neg b,\\ &\neg p_{(x+1<0)} \vee a,\\ &\neg b \vee \neg p_{(y=1)}\}\end{aligned}$$

It is not hard to see that φ^p is satisfiable. Suppose that the function pick_total_assignment(A^p,φ^p) returns us the following Boolean model for φ^p:

$$\mu^p = \{p_{(x>0)}, a, \neg b, p_{(x+1<0)}, \neg p_{(y=1)}\}$$

Now we need to check the T-satisfiability of $B2T(\mu^p)$. Since we are interested in checking the consistency of the current Boolean assignment with theory T, here we only need to take into account the literals corresponding to the theory, i.e., we have to check the T-satisfiability of $\{x > 0, x + 1 < 0, \neg(y = 1)\}$. This is obviously T-unsatisfiable, so we get a subset of T-inconsistent literals from the T-solver, e.g., $\pi = \{x > 0, x + 1 < 0\}$, and we extend φ^p with the learned clause, namely $\neg p_{(x>0)} \vee \neg p_{(x+1<0)}$. Then the search starts again.

In practice, the enumeration of Boolean models is carried out by means of efficient implementations of the DPLL algorithm [ZM02], where the partial assignments μ^p are

incrementally built. These systems inherit the spectacular progress in performance from SAT solvers in the last decade, achieved thanks to better implementation techniques and conceptual enhancements, which have been partially described in Chapter 3. Adaptations of SAT techniques to the SMT framework have been described in [SS06]. Unit propagation (see Section 3.2.2) is used extensively to perform all the assignments which derive deterministically from the current partial truth assignment μ^p. This allows the system to prune the search space and to backtrack as high as possible in the search tree (see Example 9).

An important improvement consists on checking the T-satisfiability of partial assignments, in order to anticipate possible conflicts. Theory deduction can be used to reduce the search space by explicitly returning truth values for unassigned literals, as well as constructing and learning implications. The deduction capability is a very important aspect of theory solvers, since getting short explanations (conflict sets) from the theory solver is essential in order to keep the learned lemmas as short as possible. Apart from saving memory space, shorter lemmas will allow for more pruning in general.

In the approach presented so far, the T-solver provides information only after a T-inconsistent partial assignment has been generated. In this sense, the T-solver is used only to validate the search a posteriori, not to guide it a priori. In order to overcome this limitation, the T-solver can also be used in a given DPLL state $M \parallel \varphi$ to detect literals l occurring in φ such that $M \models_T l$, allowing the DPLL procedure to move to the state $Ml \parallel \varphi$. This is called theory propagation.

Finally, as it happens in SAT solving, in order to avoid getting stuck in hard portions of the search space, most SMT systems periodically do restarts with the hope of exploring easier successful branches.

4.3 Abstract DPLL Modulo Theories

In [NOT06] we can found the adaptation to SMT of the abstract DPLL procedure (see Section 3.2.4) where the $Learn$, $Forget$ and $Backjump$ rules are slightly modified, such

that entailment between formulas now becomes entailment in T:

T-*Backjump* :

$$Ml^dN \parallel \varphi, C \implies Ml' \parallel \varphi, C \quad \text{if} \quad \begin{cases} Ml^dN \models \neg C \text{ and there is} \\ \text{some clause } C' \vee l' \text{ such that:} \\ \varphi, C \models_T C' \vee l' \text{ and } M \models \neg C', \\ l' \text{ is undefined in } M, \text{ and} \\ l' \text{ or } \neg l' \text{ occurs in } \varphi \text{ or in } Ml^dN. \end{cases}$$

T-*Learn* :

$$M \parallel \varphi \implies M \parallel \varphi, C \quad \text{if} \quad \begin{cases} \text{all atoms of } C \text{ occur in } \varphi \text{ or in } M \text{ and} \\ \varphi \models_T C. \end{cases}$$

T-*Forget* :

$$M \parallel \varphi, C \implies M \parallel \varphi \quad \text{if} \quad \varphi \models_T C.$$

Example 17 *Let us consider the following SMT formula φ:*

$$x_1 \wedge g(a) = c \wedge (f(g(a)) \neq f(c) \vee g(a) = d) \wedge c \neq d$$

Here the theory involved is Equality and Uninterpreted Functions. *In this theory functions do not have any predefined interpretation. However, equality axioms state that, independently from what is supposed to compute, a function always returns equal values for equal arguments, i.e., $a = b \rightarrow f(a) = f(b)$.*

The first step is to purify the formula, i.e., mapping propositional variables to themselves, and ground atoms into fresh propositional variables:

$$\begin{array}{lcl} x_2 & \leftrightarrow & g(a) = c \\ x_3 & \leftrightarrow & f(g(a)) \neq f(c) \\ x_4 & \leftrightarrow & g(a) = d \\ x_5 & \leftrightarrow & c \neq d \end{array}$$

The resulting Boolean formula φ^p is:

$$x_1 \wedge x_2 \wedge (x_3 \vee x_4) \wedge x_5$$

A possible sequence of applications of the DPLL rules could be the following:

$$
\begin{array}{lcll}
\emptyset & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{UnitPropagate} \\
x_1 & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{UnitPropagate} \\
x_1 x_2 & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{UnitPropagate} \\
x_1 x_2 x_5 & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{Decide} \\
x_1 x_2 x_5 x_3^d & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{T-Backjump (T-inconsistency)} \\
x_1 x_2 x_5 \neg x_3 & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{UnitPropagate} \\
x_1 x_2 x_5 \neg x_3 x_4 & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{Fail (T-inconsistency)}
\end{array}
$$

In order to model theory propagation we require the following rule:

$$
\begin{array}{l}
\mathit{TheoryPropagate}: \\
M \parallel \varphi \implies M l \parallel \varphi \quad \text{if} \quad \begin{cases} M \models_T l, \\ l \text{ or } \neg l \text{ occurs in } \varphi, \text{ and} \\ l \text{ is undefined in } M. \end{cases}
\end{array}
$$

Example 18 *Evolution of Example 17 with theory propagation:*

$$
\begin{array}{lcll}
\emptyset & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{UnitPropagate} \\
x_1 & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{UnitPropagate} \\
x_1 x_2 & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{TheoryPropagate} \\
x_1 x_2 \neg x_3 & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{UnitPropagate} \\
x_1 x_2 \neg x_3 x_4 & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{TheoryPropagate} \\
x_1 x_2 \neg x_3 x_4 \neg x_5 & \| & x_1, x_2, x_3 \vee x_4, x_5 & \Rightarrow \quad \textit{Fail}
\end{array}
$$

In this case, no backjumping has been necessary.

4.4 Theories and Logics

The Satisfiability Modulo Theories Library (SMT-LIB) [BST10a] has the goal of establishing a library of benchmarks for SMT, as well as to establish a common standard for the specification of benchmarks and of background theories. The Satisfiability Modulo Theories Competition (SMT-COMP) is an associated yearly competition for SMT solvers. Among the logics considered in the SMT-LIB there are:

- QF_UF: the quantifier-free fragment of first order logic with equality and no restrictions on the signature (hence the name UF for Uninterpreted Functions). The theory of *Equality and Uninterpreted Functions* (EUF, or simply UF) is also known as the empty theory, as far as we are concerned with first order logic with equality (i.e., with equality built-in). If Γ is a set of equalities and Δ is a set of inequalities, then the satisfiability of $\Gamma \cup \Delta$ in QF_UF can be determined as follows [NO80]:

  - Let τ be the set of terms appearing in $\Gamma \cup \Delta$.
  - Let $\sim$ be the equivalence relation on τ induced by Γ i.e., its closure under reflexivity, symmetry and transitivity.
  - Let $\sim^*$ be the congruence closure of $\sim$, obtained by closing $\sim$ with respect to the congruence axiom $\overline{s} = \overline{t} \rightarrow f(\overline{s}) = f(\overline{t})$, where $\overline{s}$ and $\overline{s}$ are vectors of symbols.
  - $\Gamma \cup \Delta$ is satisfiable iff for each $s \neq t \in \Delta, s \not\sim^* t$.

  It is possible (and sometimes preferable) to eliminate all uninterpreted function symbols by means of Ackermann's reduction [Ack68]. In Ackermann's reduction, each application $f(a)$ is replaced by a variable f_a, and for each pair of applications $f(a)$, $f(b)$ the formula $a = b \rightarrow f_a = f_b$ is added, i.e., the single theory axiom $x = y \rightarrow f(x) = f(y)$ of the theory becomes instantiated as necessary.

- *Linear Arithmetic* over the integers (QF_LIA) or the reals (QF_LRA). Closed quantifier-free formulas with Boolean combinations of inequations between linear polynomials over integer (real) variables, e.g., $(3x + 4y \geq 7) \rightarrow (z = 3)$ where x, y and z are integer variables. These inequalities can be placed in a normal form $c_0 + \sum_{i=1}^{n} c_i * x_i \leq 0$, where each c_i is a rational constant and the variables x_i are integer (real) variables. The most common approach for solving linear arithmetic is the Simplex method for real variables and the MILP for integer variables (see Subsection 2.7). A description of a QF_LIA and a QF_LRA solver can be found in [DdM06a].

- *Difference Logic* over the integers (QF_IDL) or the reals (QF_RDL). Fragment of linear arithmetic in which arithmetic atoms are restricted to have the form $x - y \bowtie k$, where x and y are numeric (integer or real) variables, k is a numeric (integer or real) constant and $\bowtie \in \{=, <, >, \leq, \geq\}$. In the usual solving method, first of all, the atoms are rewritten in terms of $\leq$. Then, the resulting atoms can be represented as a weighted directed graph with variables as vertices and edges from x to y labeled with k for every atom $x - y \leq k$. A formula is unsatisfiable iff there exists a path $x_1 \xrightarrow{k_1} x_2 \ldots x_n \xrightarrow{k_n} x_1$ such that $k_1 + k_2 + \cdots + k_n < 0$. A description of a QF_RDL solver can be found in [NO05].

- *Non-linear Arithmetic* over the integers (QF_NIA) or the reals (QF_NRA). Quantifier free integer or real arithmetic with no linearity restrictions, i.e., with clauses like $(3xy > 2 + z^2) \vee (3xy = 9)$ where x, y and z are variables. A possible technique to check the satisfiability of these formulas is to transform the problem into a linear approximation [BLO+12].

- *Arrays* (QF_AX). Closed quantifier-free formulas over the theory of arrays with extensionality. The signature of this theory consists of two interpreted function

symbols: $read$, used to retrieve the element stored at a certain index of the array, and $write$, used to modify an array by updating the element stored at a certain index. The axioms of the theory of arrays are:

$$\begin{array}{l} \forall a : Array, \forall i, j : Index, \forall x : Value \\ \quad i = j \quad \Rightarrow \quad read(write(a, i, x), j) = x \\ \quad i \neq j \quad \Rightarrow \quad read(write(a, i, x), j) = read(a, j) \end{array}$$

Finally, the extensionality axiom states that two arrays that coincide on all indices are indeed equal:

$$\begin{array}{l} \forall a, b : Array \\ \quad (\forall i : Index \;\; read(a, i) = read(b, i)) \quad \Rightarrow \quad a = b \end{array}$$

A possible approach to decide the satisfiability of ground literals in this theory is to transfer the atoms to the Equality and Uninterpreted Functions theory. Other approaches are based on a careful analysis of the problem that allows to infer, for each array, which are the relevant indices and which values are stored at these indices of the array [SBDL01, BNO+08a].

- *Bit vectors* (QF_BV). Closed quantifier-free formulas over the theory of fixed-size bit vectors. Bit vectors are normally used for representing memory contents. Common operations are: extraction of a sequence of bits, concatenation, arithmetic operations ($+$, $-$, $*$, ...), bit-wise operations (and, or, not, ...), etc. The state-of-the-art methods for checking the satisfiability of a given bit vector formula are based on reduction to SAT (bit-blasting). Each bit vector is encoded into a set of Boolean variables and the operators are encoded into logical circuits [BKO+07].

- *Other theories*. In the literature we can find some other theories of interest not considered in the SMT-LIB, such as Alldifferent [BM10] and the teory of costs [CFG+10].

The expressivity of each of these logics has its corresponding computational price. Checking consistency of a set of IDL constraints has polynomial time complexity whereas checking consistency of a set of LIA constraints is NP-complete. The non-linear case is in general (with no bounds on the domains of the variables) undecidable.

4.4.1 Combination of Theories

Many natural SMT problems contain atoms from multiple theories. When dealing with two or more theories, a standard approach is to handle the integration of the different theories by performing some sort of search on the equalities between their shared (or *interface*)

variables. First of all, formulas are purified by replacing terms with fresh variables, so that each literal only contains symbols belonging to one theory. For example,

$$a(1) = x + 2$$

is translated into

$$\begin{aligned} a(v_1) &= v_2 \\ v_1 &= 1 \\ v_2 &= x + 2 \end{aligned}$$

where the first literal belongs to UF, and the last two to LIA. Variables v_1 and v_2 are then called *interface variables*, as they appear in literals belonging to different theories. An *interface equality* is an equality between two interface variables. All theory combination schemata, e.g., Nelson-Oppen [NO79], Shostak [Sho84], or Delayed Theory Combination (DTC) [BBC$^+$06], rely to some point on checking equality between interface variables, in order to ensure mutual consistency between theories. This may imply to assign a truth value to all the interface equalities. Since the number of interface equalities is given by $|\mathcal{V}| \cdot (|\mathcal{V}| - 1)/2$, where $|\mathcal{V}|$ is the number of interface variables, the search space may be enlarged in a quadratic factor in the number of interface variables.

In the case of combining UF with another theory T, an alternative approach is to eliminate the uninterpreted function symbols by means of Ackermann's reduction [Ack68], and then solving the resulting SMT problem only with theory T. However, this has the same disadvantage as theory combination since the number of additional literals is quadratic in the size of the input and, in fact, as shown in [BCF$^+$06], there is no clear winner between DTC and Ackermannization.

4.5 MaxSMT and Weighted SMT

We can adapt the concept of MaxSAT and Weighted SAT for over-constrained SMT problems, i.e., SMT formulas such that it is not possible to satisfy all clauses at the same time. A MaxSMT instance is an SMT instance where clauses may have an associated weight (cost) of falsification.

Definition 4.5.1 *Given an SMT formula, the maximum SMT (MaxSMT) problem consists in finding a solution that maximizes the number of satisfied SMT clauses.*

A Partial MaxSMT problem is a MaxSMT problem where there is a set of clauses that can be violated (soft clauses) and a set of clauses that must be satisfied (hard clauses).

The objective is then to minimize the number of falsified clauses, while satisfying all hard clauses.

As in SAT, an SMT formula is considered to be a set of clauses while, in (Partial) MaxSMT, a CNF formula is considered to be a multiset of clauses, because repeated clauses cannot be collapsed into a unique clause. More formally:

Definition 4.5.2 *A* Weighted SMT clause[2] *is a pair* (C, w)*, where* C *is an SMT clause and* w *is a natural number or infinity (indicating the penalty for violating* C*). A* Weighted (Partial) MaxSMT formula *is a multiset of Weighted SMT clauses*

$$\varphi = \{(C_1, w_1), \ldots, (C_m, w_m), (C_{m+1}, \infty), \ldots, (C_{m+m'}, \infty)\}$$

where the first m *clauses are soft and the last* m' *clauses are hard.*

If there are no hard clauses (clauses with infinite cost of falsification) we speak of Weighted SMT, and if there are we talk about Weighted Partial SMT.

The optimal cost of a formula is the minimal cost of all its assignments. An optimal assignment is an assignment with optimal cost.

Definition 4.5.3 *The Weighted (Partial) SMT problem for a Weighted (Partial) SMT formula is the problem of finding an optimal assignment for that formula.*

4.6 Lazy Clause Generation

Many authors consider the Lazy Clause Generation (Lazy_fd), see [OSC09, FS09], CSP solving method as an SMT solver. In fact, in both of them a SAT solver and a Theory Solver coexist. In the case of Lazy_fd the theory solver always is a finite domain solver with propagators.

But there exist many differences between SMT and Lazy_fd. In the initial Lazy_fd approach [OSC09] (see the scheme in Figure 4.1) all integer variables X with initial domain $D(X) = \{l, \ldots, u\}$ are encoded into SAT using a regular encoding (see Subsection 3.5.2). Each value $i \in D(X)$ is encoded using two variables x_i and $x_i^{\leq}$ (for $i = u$ this latter is not necessary). In order to prevent inconsistent assignments to these Boolean variables (for example x_3 and $x_2^{\leq}$ to $true$) Lazy_fd adds to the Boolean formula in the SAT solver the clauses that define the conditions between them, as described in Subsection 3.5.2.

[2] In fact these could be general SMT formulas, not necessarily disjunctions of atoms.

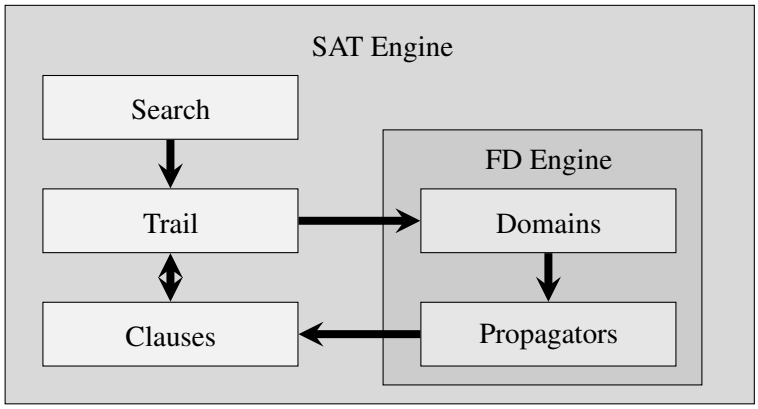


Figure 4.1: Old architecture of Lazy_fd

In Lazy_fd, the propagator functions (see Subsection 2.4.3) are implemented by adding clauses to the Boolean formula, describing each propagation lazily. Instead of applying propagator f to domain $\mathcal{D}$ to obtain $f(\mathcal{D})$, whenever $f(\mathcal{D}) \subset \mathcal{D}$ we build a clause that encodes the change in the domains. Similarly when $f(D(X))$ is empty the propagator must create a clause that explains the failure. The explaining clauses of the propagation are added to the Boolean formula in the SAT solver, on which unit propagation is performed. Since the clauses will always have the form $C_t \rightarrow l$, where C_t is a conjunction of literals that are true in the current assignment, and l is a literal that is not true in the current assignment, the newly added clause will always cause unit propagation, adding l to the current assignment.

Note that in the presented lazy clause generation solver, the search is controlled by the SAT engine. After making a decision, unit propagation is performed to reach a unit propagation fix point. Every fixed literal is then translated into a domain change and the appropriate propagators are woken up. An important aspect is that when a propagator f such that $f(\mathcal{D}) \neq \mathcal{D}$ is found, the propagator does not directly modify the domain $\mathcal{D}$ but instead generates a set of clauses which explain the domain changes. Each clause is passed to the SAT solver, starting a new round of unit propagations. This continues until a fixed point is reached and a new SAT decision is made, or a contradiction is found and a backjumping step is performed.

Example 19 *Consider* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *where* $\mathcal{X} = \{X, Y, Z\}, D(X) = D(Y) = D(Z) = \{1, 2, 3\}$ *and* $\mathcal{C} = \{C_1 = (X = Y), C_2 = (Y < Z)\}$.

The propagators are:

$$f_{C_1}(\mathcal{D}) = \mathcal{D}' \quad \textit{where} \quad \begin{cases} D'(X) = D(X) \cap D(Y) \\ D'(Y) = D(Y) \cap D(X) \\ D'(Z) = D(Z) \end{cases}$$

$$f_{C_2}(\mathcal{D}) = \mathcal{D}' \quad \textit{where} \quad \begin{cases} D'(X) = D(X) \\ D'(Y) = \{d \in D(Y) \mid d < max(D(Z))\} \\ D'(Z) = \{d \in D(Z) \mid d > min(D(Y))\} \end{cases}$$

The initial clauses for the regular encoding of the domains are:

- *Variables:*

$$\{x_1, x_2, x_3, x_{\overline{1}}^{\leq}, x_{\overline{2}}^{\leq}, y_1, y_2, y_3, y_{\overline{1}}^{\leq}, y_{\overline{2}}^{\leq}, z_1, z_2, z_3, z_{\overline{1}}^{\leq}, z_{\overline{2}}^{\leq}\}$$

- *Clauses:*

$$\begin{aligned} \{&\neg x_{\overline{1}}^{\leq} \vee x_{\overline{2}}^{\leq}, \neg x_1 \vee x_{\overline{1}}^{\leq}, \neg x_2 \vee x_{\overline{2}}^{\leq}, \neg x_2 \vee \neg x_{\overline{1}}^{\leq}, \\ &\neg x_3 \vee \neg x_{\overline{2}}^{\leq}, x_1 \vee \neg x_{\overline{1}}^{\leq}, \neg x_{\overline{2}}^{\leq} \vee x_{\overline{1}}^{\leq} \vee x_2, x_{\overline{2}}^{\leq} \vee x_3, \\ &\neg y_{\overline{1}}^{\leq} \vee y_{\overline{2}}^{\leq}, \neg y_1 \vee y_{\overline{1}}^{\leq}, \neg y_2 \vee y_{\overline{2}}^{\leq}, \neg y_2 \vee \neg y_{\overline{1}}^{\leq}, \\ &\neg y_3 \vee \neg y_{\overline{2}}^{\leq}, y_1 \vee \neg y_{\overline{1}}^{\leq}, \neg y_{\overline{2}}^{\leq} \vee y_{\overline{1}}^{\leq} \vee y_2, y_{\overline{2}}^{\leq} \vee y_3, \\ &\neg z_{\overline{1}}^{\leq} \vee z_{\overline{2}}^{\leq}, \neg z_1 \vee z_{\overline{1}}^{\leq}, \neg z_2 \vee z_{\overline{2}}^{\leq}, \neg z_2 \vee \neg z_{\overline{1}}^{\leq}, \\ &\neg z_3 \vee \neg z_{\overline{2}}^{\leq}, z_1 \vee \neg z_{\overline{1}}^{\leq}, \neg z_{\overline{2}}^{\leq} \vee z_{\overline{1}}^{\leq} \vee z_2, z_{\overline{2}}^{\leq} \vee z_3\} \end{aligned}$$

The process of the lazy clause generation solver start when the SAT solver takes a decision, for example adding x_2 to the current truth assignment. At this moment, thanks to unit propagation, the domain of variable X becomes $\{2\}$. This is handled by the finite domain solver which activates the f_{C_1} propagator, generating the clause $\neg x_2 \vee y_2$. In turns, unit propagation on this lastly added clause changes the domain of Y to $\{2\}$. This is also captured by the finite domain solver, which activates the f_{C_2} propagator, generating a new clause. As said, the process continues until a fixed point is reached and a new SAT decision is made, or a contradiction is found and a backjumping step is performed.

There is a new version of Lazy_fd [FS09] (see the scheme in Figure 4.2). In this approach the SAT solver is not the core engine, and acts only as an oracle to control the possible values of the variable domains. This framework is very different to the standard SMT solvers' scheme.

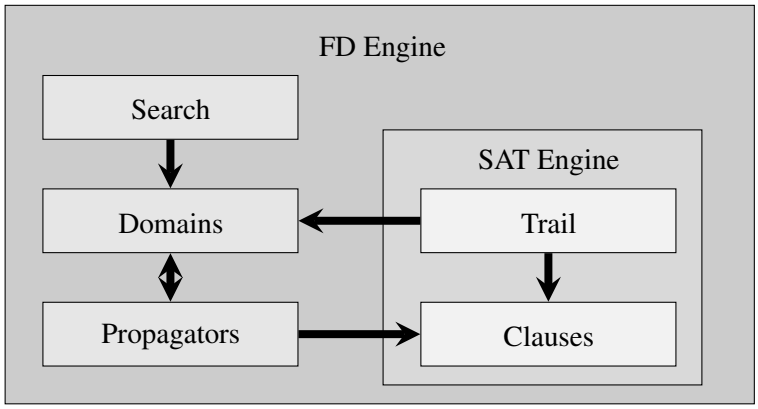


Figure 4.2: New architecture of Lazy_fd

The search is controlled by the finite domain solver. When a propagator f is executed updating a variable domain ($f(\mathcal{D}) \neq \mathcal{D}$) or causing a failure ($\mathcal{D}$ is empty) it posts an explanation clause to the SAT solver that explains the domain reduction or the failure.

This method does not initially need all the clauses of the regular encoding of the variable domains. Instead, they can be lazily introduced. There are two possible ways of doing this:

- **Array Encoding**. In this encoding there exist two arrays of Boolean variables:
  - The array of inequality (regular) literals $x_i^{\leq}, i \in [l, u-1]$, which are generated eagerly with the restrictions $\bigwedge_{i=1}^{i<n-1} \neg x_i^{\leq} \vee x_{i+1}^{\leq}$.
  - The array of equality literals $x_i, i \in [l, u]$, which are generated lazily. When an equality literal x_i is necessary (occurs in a propagator or in a model), the following clauses are posted:

$$\neg x_i \vee x_i^{\leq}$$
$$\neg x_i \vee \neg x_{i-1}^{\leq}$$
$$x_{i-1}^{\leq} \vee \neg x_i^{\leq} \vee x_i$$

- **List Encoding**. In this encoding the inequality and the equality literals are generated lazily when they are required for propagation or explanation.

  When an inequality literal $x_i^{\leq}$ is introduced:

  - The closest existing bounds for x_i are determined:

$$l = \max\{j \mid x_j^{\leq} \text{ exists}, j < i\}$$
$$u = \min\{j \mid x_j^{\leq} \text{ exists}, i < j\}$$

– The following new domain clauses are posted:

$$\neg x_l^{\leq} \vee x_i^{\leq}$$
$$\neg x_i^{\leq} \vee x_u^{\leq}$$

When an equality literal x_i is introduced, the introduction of the literals $x_i^{\leq}$ and $x_{i-1}^{\leq}$ is required. Then the solver proceeds as in the case of array encoding.

The clauses generated by the propagators are equal to the ones of the initial Lazy_fd the propagators. However, this new scheme allows search flexibility, since it is the finite domain and not the SAT solver that controls the search and uses less memory resources.

Example 20 *Continuing with Example 19, the initial clauses for the array encoding would be:*

- *Variables:*
$$\{x_1^{\leq}, x_2^{\leq}, y_1^{\leq}, y_2^{\leq}, z_1^{\leq}, z_2^{\leq}\}$$
- *Clauses:*
$$\{\neg x_1^{\leq} \vee x_2^{\leq}, \neg y_1^{\leq} \vee y_2^{\leq}, \neg z_1^{\leq} \vee z_2^{\leq}\}$$

The process of the lazy clause generation solver starts when the search algorithm tests a label, for example X with value 2. In this moment it has to generate the x_2 equality variable and the clauses $\neg x_2 \vee x_2^{\leq}$, $\neg x_2 \vee \neg x_1^{\leq}$, $x_1^{\leq} \vee \neg x_2^{\leq} \vee x_2$. Then it activates the f_{C_1} propagator, which generates the clause $\neg x_2 \vee y_2$. As y_2 is not present in the Boolean formula, it has to generate this equality variable and the associated clauses. This process continues until a fixed point is reached. Then the search algorithm tests a new label or does brackjumping.

Lazy_fd is a highly competitive approach in general, and its last implementation is one of the most competitive of state-of-the-art CSP solvers.

Chapter 5

Encoding CSP into SMT

In this chapter we begin with the contributions of the thesis. As we have said, one of the main objectives of the thesis is to show the robustness of SMT solvers to solve CSPs, in particular, by encoding CSPs into SMT formulas. This chapter will take into account the concepts explained in previous chapters (CSP in Chapter 2, SAT in Chapter 3 and SMT in Chapter 4) to get a correct, complete and efficient encoding. In the encodings we consider several theories: difference logic, linear arithmetic, and equality and uninterpreted functions. This chapter begins with the state-of-the-art of systems based on translation from CSP into SAT and SMT. It continues with the two systems developed in the context of this thesis, translating CSPs into SMT formulas: `Simply` and `fzn2smt`. In some sense, as SMT can be seen as a natural evolution from SAT, the two systems presented can be seen as a natural evolution from other systems based on encodings to SAT. In the description of `fzn2smt` we make special emphasis on the solutions to optimization problems. We also provide an extensive interpretation of the results obtained in our experiments and, in particular, we study the impact of the Boolean component of the instances in the performance of `fzn2smt`.

The content of this chapter is part of our publications:

- *`Simply`: a Compiler from a CSP Modeling Language to the SMT-LIB Format* [BPSV09],
- *A System for Solving Constraint Satisfaction Problems with SMT* [BSV10], and
- *Solving constraint satisfaction problems with SAT modulo theories* [BPSV12].

The objectives of the thesis achieved in this chapter are the first one (to show that SMT can be a good generic solving approach for CSP) and partially the second (to prove that using an SMT solver in conjunction with appropriate algorithms can be a robust approach for optimization variants of CSP). The contributions described in this chapter range from the first to the fourth one.

5.1 State-of-the-Art

In the last decade there have been important advances in SAT solving techniques, to the point that SAT solvers have become a viable engine for solving constraint satisfaction problems [Wal00, CMP06, TTKB09]. On the other hand, SAT techniques have been adapted for more expressive logics resulting in new techniques such as SMT. Although most SMT solvers are restricted to decidable quantifier free fragments of their logics, this suffices for many applications. In fact, there are already promising results in the direction of adapting SMT techniques for solving CSPs, even in the case of combinatorial optimization [NO06]. Fundamental challenges for SMT with respect to constraint programming and optimization are suggested in [NORCR07].

In the literature there are many attempts, more or less generic, to translate CSP into SAT. They are based on different encodings, some of them explained in Section 3.5. For example we can

- SUGAR. This is a SAT-based constraint solver that uses the SAT-encoding method named order encoding. Sugar became a winner in GLOBAL categories of 2008 and 2009 CSP Solver Competitions. It can solve COP and MaxCSP. See [TTKB09, Sug11].
- CSP2SAT4J. This system is part of the Sat4j (the Boolean satisfaction and optimization library for Java) and translates given CSPs in extension into SAT problems. It uses direct encodings and support encoding. See [Sat05].
- FznTini. It solves constraint satisfaction and optimization problems written in FlatZinc (not involving floating point numbers) via Booleanization and calls to the Tinisat SAT solver. It uses the SAT-encoding method named Log Encoding. See [Hua08, Fzn08]
- SPEC2SAT. Transforms problem specifications written in NP-SPEC [CIP+00] (a logic-based specification language which allows us to specify combinatorial problems in a declarative way) into SAT instances. It uses the direct encoding. See [CS05, Spe05].
- BEE. A compiler which enables to encode finite domain constraint problems to CNF. During compilation, BEE applies optimizations such as equi-propagation, partial-evaluation, and a careful selection of encoding techniques per constraint. It basically uses the order encoding. See [MC12, Bee12].

Nevertheless, until the systems presented in this thesis, there has been no attempt to make a generic translation of CSP into SMT. In this work we solve this gap, and we

get two very efficient generic systems. Note that our systems use SMT solvers as black boxes, hence we believe that there is much room for improvement with respect to CSP solving via SMT.

Recently there has been another approach to make a generic translation of CSPs into SMT formulas using an imperative style in constraints posting [MJ10]. Instead of using linear integer arithmetic as we do, the system URBiVA uses Bit Vectors. This work, and its results, are not easily comparable with ours because of the different nature of the modelling languages.

5.2 Simply

To have the possibility of modelling CSPs and solving them via SMT we developed `Simply`. `Simply` is intended to be a declarative programming system for easy modelling and solving of CSPs. Although the richness of its input language does not reach the level of ESSENCE [FHJ+08] or MiniZinc [Nic07, G1210], its simplicity makes it really practical. The input language of `Simply` (see Fig. 5.1) is similar to that of EaCL [MTW+99] and MiniZinc, and its main implemented features are arrays, *Forall* sentences, list comprehensions, and some global constraints. `Simply` works in the spirit of SPEC2SAT, which transforms problem specifications written in NP-SPEC into SAT instances. However, as said, the input language of `Simply` is similar to that of EaCL and, most importantly, it generates SMT instances according to the standard SMT-LIB language [RT06] instead of SAT instances. Then, the problem can be solved by using any SMT solver supporting the required theories and the SMT-LIB 1.2 language.

```
Problem:queens_8
    Data
        n:=8;
    Domains
        Dom rows=[1..n];
    Variables
        IntVar q[n]::rows;
    Constraints
        AllDifferent([q[i] | i in [1..n]]);
        Forall(i in [1..n-1]) {
            Forall(j in [i+1..n]) {
                q[i]-q[j]<>j-i;
                q[j]-q[i]<>j-i;
            }
        }
```

Figure 5.1: A Simply encoding for the 8-Queens problem.

In Figure 5.1 we can see an example of a `Simply` instance for the n-*Queens* problem: given a number n of queens of the chess game, the problem is to find a position in a board of size $n \times n$, for each queen, such that no queen threatens any other (i.e., they are not in the same column, row or diagonal). The array `q` contains `n` integer variables where `q[i]`, for `i` in `1..n`, denotes the row where the queen of column `i` is placed. The problem is solvable iff there exists an assignment for `q`, according to its domain, such that all the posted constraints are satisfied. In this case we require first, that all the values of `q` are different, i.e., no two queens are in the same row (constraint `Alldifferent` in the instance) and second, that the distances between the indexes (columns) and values (rows) of any two pair of elements of the array are distinct, i.e., no two queens are in the same diagonal (constraint induced by the two *Forall* statements in the instance).

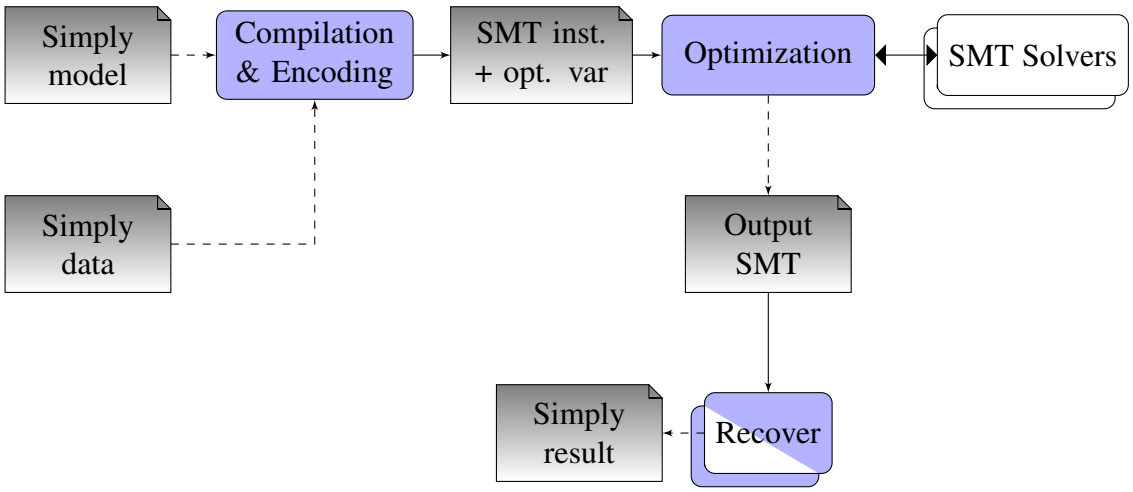


Figure 5.2: The architecture of Simply.

In Figure 5.2 we can see the architecture of `Simply`. Let the input of the compiler be the files (model and data). In the compilation process all constants are replaced by their associated value, and all variables are translated into SMT integer variables. Constraining a variable to its domain results in a disjunction of equalities when the domain is an explicit enumeration of values, or into a conjunction of two inequality predicates when the domain is described as a range. The translation of the constraints typically results into a conjunction of QF_LIA predicates and, in some occasions (as in this case for *8-Queens*), into a conjunction of QF_IDL predicates. In the end, the compilation process produces an SMT file (see Fig. 5.3) in the standard SMT-LIB format.

The generated SMT problem instance can then be solved by any of the SMT solvers supporting the QF_LIA logic (Yices, Z3, Barcelogic,...). Solving the SMT instance with the desired SMT solver will result into a *sat* or *unsat* answer (notice that these solvers are complete). In addition, some of them (e.g., Yices [DdM06b]) can return a model (in

```
(benchmark queens_8.smt
  :source {Generated by Simpl.y, ima.udg.edu (LaP)}
  :category {testing}
  :logic QF_IDL
  :extrafuns ((q_1 Int) ..... (q_8 Int))
  :formula
 (and
  (and
   (and (>= q_1 1) (<= q_1 8))
    ...
   (and (>= q_8 1) (<= q_8 8))
  )
  (and
   (distinct  q_1 ... q_8)
   (and
     (and (distinct (- q_1 q_2) (- 2 1))
          (distinct (- q_2 q_1) (- 2 1))
           ...
          (distinct (- q_1 q_8) (- 8 1))
          (distinct (- q_8 q_1) (- 8 1)) )
     (and (distinct (- q_2 q_3) (- 3 2))
          (distinct (- q_3 q_2) (- 3 2))
           ...
          (distinct (- q_2 q_8) (- 8 2))
          (distinct (- q_8 q_2) (- 8 2)) )
      ...
     (and (distinct (- q_7 q_8) (- 8 7))
          (distinct (- q_8 q_7) (- 8 7)) )
   )
  )
 )
)
```

Figure 5.3: SMT problem resulting from the compilation to the 8-Queens instance.

```
sat (= q_1 5)(= q_2 2)(= q_3 8)(= q_4 1)
    (= q_5 4)(= q_6 7)(= q_7 3)(= q_8 6)
```

Figure 5.4: The answer of Yices to the 8-Queens instance.

particular, as shown in Fig. 5.4, the values of the SMT variables) when the problem is satisfiable. In general the names of the variables are easy to interpret. However, we have a recovering process from SMT solutions to values of the variables in the original file.

Optimization is not supported in the SMT-LIB language but `Simply` deals with COPs by means of iterative calls performing a binary search on the domain of the variable

to optimize.

5.2.1 Structure of Simply

Since we are interested in modelling CSPs easily, one of the goals of our tool is simplicity. For this reason, we have chosen the input language of `Simply` to be similar to that of EaCL and MiniZinc. In Figures 5.5 and 5.6 we can see the syntax of our modelling language and in Figure 5.7 the syntax of the data file. The tool, as well as a technical report and other information are available at: `http://ima.udg.edu/Recerca/lap/simply`.

A CSP instance specification in our language has four parts:

1. `Data` definition. In this part is where the *constants* that will be used in the rest of the specification are defined. These can be either integer or Boolean constants and vectors of integers associated to a list. In all cases their associated expressions must be valuable at compilation time.

2. `Domains` definition. A *domain* characterizes the set of possible values of a variable. It can be defined by a list with ranges and individual values, or by a list comprehension. The values of these lists must be valuable at compilation time.

3. `Variables` declaration. A *variable* can denote either an integer, a Boolean, or a multidimensional array of integers or of Booleans. Integer variables (and elements of integer arrays) are in fact *finite domain variables* and, hence, they must be constrained to some previously defined finite domain.

4. `Constraints` posting. In this part is where the problem is modeled by posting the *set of constraints* that define the feasible solutions of the problem.

When the instance is given in two files (the model and the data files) the data file overwrites the `Data` definition part of the model. Thanks to this duality of files, we can model the problem once, in the model file, and easily create as many instances as we want by simply setting the values of the instances in the data files.

5.2.2 Constraints

The input language deals with *formulas*, *global constraints* and the `ite` constraint.

- Formulas, or Boolean expressions, are basic constraints built up from integer and Boolean variables and constants. The following operators are supported: =, <>, <, =< and >= for integers and `Not`, `And`, `Or`, `Xor`, `Implies` and `Iff` for Booleans.

⟨*simply_program*⟩ ::= `Problem:`⟨*id*⟩ ⟨*data*⟩ ⟨*domains*⟩ ⟨*variables*⟩ ⟨*user_defined_functions*⟩? ⟨*constraints*⟩ ⟨*optimitzation*⟩?

⟨*data*⟩ ::= `Data` ⟨*data_def*⟩*

⟨*data_def*⟩ ::= `int` ⟨*id*⟩ = ⟨*arithm_exp*⟩ `;`
| `bool` ⟨*id*⟩ = ⟨*formula*⟩ `;`
| (`bool` | `int`) ⟨*array_id*⟩ = ⟨*list*⟩ `;`

⟨*domains*⟩ ::= `Domains` ⟨*domain_def*⟩*

⟨*domain_def*⟩ ::= `Dom` ⟨*id*⟩ = ⟨*list*⟩ `;`

⟨*variables*⟩ ::= `Variables` ⟨*variable_def*⟩*

⟨*variable_def*⟩ ::= `IntVar` ⟨*array_id*⟩ (`,` ⟨*array_id*⟩)* `::` (⟨*id*⟩ | `int`) `;`
| `BoolVar` ⟨*array_id*⟩ (`,` ⟨*array_id*⟩)* `;`

⟨*array_id*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ `[` ⟨*arithm_exp*⟩ (`,` ⟨*arithm_exp*⟩)* `]`

⟨*user_defined_functions*⟩ ::= `Functions` ⟨*user_defined_functions_def*⟩*

⟨*user_defined_functions_def*⟩ ::= ⟨*def_constraint*⟩
| ⟨*def_function*⟩ `;`

⟨*def_constraint*⟩ ::= `DefConstraint` ⟨*id*⟩ ⟨*dfc_params*⟩ { ⟨*sentence*⟩+ }

⟨*def_function*⟩ ::= `Function` (`int` | `bool`) ⟨*id*⟩ ⟨*dfc_params*⟩ ⟨*sentence*⟩

⟨*dfc_params*⟩ ::= `(` (`int` | `bool`) ⟨*id*⟩ (`,` (`int` | `bool`) ⟨*id*⟩)* `)`

⟨*constraints*⟩ ::= `Constraints` ⟨*sentence*⟩+

⟨*sentence*⟩ ::= ⟨*constraint*⟩ `;`
| ⟨*list*⟩ `;`

⟨*constraint*⟩ ::= ⟨*formula*⟩
| ⟨*global_constraint*⟩

⟨*formula*⟩ ::= `Not` ⟨*formula*⟩
| ⟨*formula*⟩ ⟨*bool_operator*⟩ ⟨*formula*⟩
| ⟨*arithm_exp*⟩ ⟨*relational_operator*⟩ ⟨*arithm_exp*⟩
| ⟨*user_defined_references*⟩
| `If_Then_Else` (⟨*formula*⟩) { ⟨*sentence*⟩+} { ⟨*sentence*⟩+}
| ⟨*statement*⟩
| `(` ⟨*formula*⟩ `)`
| ⟨*array_id*⟩
| `True`
| `False`

⟨*user_defined_references*⟩ ::= ⟨*id*⟩ `(` ⟨*list*⟩ `)`

⟨*relational_operator*⟩ ::= = | <> | < | > | =< | >=

⟨*bool_operator*⟩ ::= `And` | `Or` | `Xor` | `Iff` | `Implies`

⟨*global_constraint*⟩ ::= `AllDifferent(` ⟨*list*⟩ `)`
| `Sum(` ⟨*list*⟩ `,` ⟨*arithm_exp*⟩ `)`
| `Count(` ⟨*list*⟩ `,` (⟨*arithm_exp*⟩|⟨*formula*⟩) `,` ⟨*arithm_exp*⟩ `)`

Figure 5.5: Simply syntax I.

- Currently the following global constraints are supported:
  - `Sum(List,Value)`. This constraint enforces equality between `Value` and the sum of all elements of `List`. When `List` is empty, `Value` is enforced to be zero.

⟨*arithm_exp*⟩	::=	⟨*numeral*⟩
	|	⟨*array_id*⟩
	|	⟨*arithm_exp*⟩ ⟨*arithm_operator*⟩ ⟨*arithm_exp*⟩
	|	`(` ⟨*arithm_exp*⟩ `)`
	|	`Abs(` ⟨*arithm_exp*⟩ `)`
	|	⟨*user_defined_references*⟩
	|	`length(` ⟨*arithm_exp*⟩ `)`
⟨*statement*⟩	::=	⟨*if_then_else*⟩
	|	⟨*forall*⟩
⟨*if_then_else*⟩	::=	`If (` ⟨*formula*⟩ `) Then {` ⟨*sentence*⟩$^+$ `}`
	|	`If (` ⟨*formula*⟩ `) Then {` ⟨*sentence*⟩$^+$ `} Else {` ⟨*sentence*⟩$^+$ `}`
⟨*forall*⟩	::=	`Forall (` (⟨*id*⟩ `in` ⟨*list*⟩ `)`$^+$ `) {` ⟨*sentence*⟩$^+$ `}`
⟨*arithm_operator*⟩	::=	`+` | `-` | `*` | `Div` | `Mod`
⟨*list*⟩	::=	`[` ⟨*list_element*⟩ (`,` ⟨*list_element*⟩)* `]`
	|	`[` (⟨*arithm_exp*⟩|⟨*formula*⟩) `|` ⟨*var_restrict*⟩(`,` ⟨*var_restrict*⟩)*`]`
⟨*list_element*⟩	::=	⟨*arithm_exp*⟩
	|	⟨*range*⟩
⟨*var_restrict*⟩	::=	⟨*id*⟩ `in` ⟨*list*⟩
	|	⟨*formula*⟩
⟨*range*⟩	::=	⟨*arithm_exp*⟩ `..` ⟨*arithm_exp*⟩
⟨*id*⟩	::=	non empty sequence of letters, digits and _, not starting with digit
⟨*numeral*⟩	::=	non empty sequence of digits
⟨*comments*⟩	::=	`%` anything else until end of line
	|	`\ *` anything else `* \`
⟨*optimitzation*⟩	::=	`Minimize` ⟨*id*⟩ `;`
	|	`Maximize` ⟨*id*⟩ `;`

Figure 5.6: Simply syntax II.

⟨*data_file*⟩	::=	⟨*data_file_def*⟩*
⟨*data_file_def*⟩	::=	⟨*id*⟩ = ⟨*arithm_exp*⟩ `;`
	|	⟨*id*⟩ = ⟨*formula*⟩ `;`
	|	⟨*id*⟩ = ⟨*list*⟩ `;`

Figure 5.7: Simply syntax of data file.

  - `Count(List,Value,N)`. This constraint states equality between `N` and the number of occurrences of `Value` in `List`. When `List` is empty, `N` is enforced to be zero.
  - `AllDifferent(List)`. This constraint requires all the elements of `List` to be different.

  Let us remark that the elements of `List`, as well as the `Value` and `N` parameters, are allowed to be arithmetic expressions containing integer variables.

- The `If_Then_Else` (ϕ) $\{$`C1`$\}$ $\{$`C2`$\}$ constraint states that, when the formula ϕ is satisfied, then the constraints `C1` must be satisfied and, when ϕ is not satisfied,

then the constraints `C2` must be satisfied. Let us remark that the formula ϕ does not need to be evaluable at compilation time.

Constraints can be posted either (i) directly, (ii) through the *If-Then-Else* statement (which has nothing to do with the `If_Then_Else` constraint), or (iii) through the *Forall* statement or list comprehensions. These last two statements are processed at compilation time. For instance, when the compiler finds an *If-Then-Else* statement, it evaluates the `If` condition. If it is true, the constraints of the `Then` branch are posted and, otherwise, the constraints of the `Else` branch are posted.

It is important to notice the difference between the *If-Then-Else* statement and the `If_Then_Else` constraint, whose condition, as said, is not evaluated at compilation time.

The semantics of a *Forall* statement is as usual and can be illustrated with the following example:

```
Forall(i in [2..4]) {m[i]<>m[i-1];}
```

The variables that define a *Forall* statement are variables that are evaluated at compilation time.

The previous *Forall* statement results into the replication of `m[i]<>m[i-1]` with the local variable `i` being replaced by the appropriate values:

```
m[2]<>m[1]; m[3]<>m[2]; m[4]<>m[3];
```

Lists can be extensional, by directly enumerating elements and ranges e.g., `[1, x, 3..5, m[a]+3]`, or intensional via *list comprehensions à la Haskell*. This powerful and expressive feature allows us to generate complicated lists easily. We illustrate its usage with the following example:

```
[ m[i,j] | i in [1..3], j in [1..3], i<>j ]
```

that results into the following list:

```
[ m[1,2], m[1,3], m[2,1], m[2,3], m[3,1], m[3,2] ]
```

The first part of a list comprehension is the *pattern*, i.e., the expression that we want to generate. Currently, patterns must be arithmetic expressions (in this example, the elements of the bidimensional array `m`). The rest of the list comprehension is formed by two distinct kinds of expressions, namely, the *generators* (in the example, `i in [1..3]` and `j in [1..3]`, that expand the pattern) and the *filters*, that restrict these expansions (e.g., `i<>j`).

5.2.3 Examples and Benchmarks

We have run the SMT solvers which participated in the QF_LIA division of the Satisfiability Modulo Theories Competition[1] (SMT-COMP) 2008, namely, Z3.2, MathSAT-4.2, CVC3-1.5, Barcelogic 1.3 and Yices 1.0.10, against some benchmarks generated with `Simply` in several known problems: Queens, BACP, Schur Lemma and Jop-shop. We have also run some solvers of different nature on the same problems, namely, G12 MiniZinc 0.9, ECLiPSe 6.0, SICStus Prolog 4.0.7, SPEC2SAT 1.1 and Comet 1.2.

The same benchmarks, after a translation from the MiniZinc modelling language to the FlatZinc low-level solver input language by using the MiniZinc to FlatZinc translator `mzn2fzn`, have been used for G12 MiniZinc 0.9, ECLiPSe 6.0 and SICStus Prolog 4.0.7. For SPEC2SAT and Comet, we have preserved as much as possible the modelling used in `Simply`. In our tests, we have used SPEC2SAT together with zChaff 2007.3.12, and with respect to Comet, only its constraint programming module has been tested. In order for the comparison to be fair, we have avoided the use of any search strategy when dealing with other solvers, as no search control is possible within SMT solvers.

Table 5.1 shows the time in seconds spent by each solver in each problem. The benchmarks were executed on a 3.00 GHz Intel® Core™ 2 Duo machine with 2 Gb of RAM running under GNU/Linux 2.6. The column labeled Simply refers to the `Simply` compilation time. The following 5 columns contain the solving time spent by the different SMT solvers on the generated SMT instances. The rest of columns detail the times (including compilation and solving) spent by solvers of other nature. Time out (t.o.) was set to 1800 seconds. Memory out is denoted by m.o.

Globally, it seems that most of the SMT solvers are good in all the problems considered. This is especially relevant if we take into account that those solvers come from the verification arena and, therefore, have not been designed with those kind of constraint problems in mind. Moreover, they seem to scale up very well with the size of the problems. Let us remark that these problems are just the first ones at hand that we have considered, i.e., we have not artificially chosen them. For this reason, SMT can be expected to provide a nice compromise between expressivity and efficiency for solving CSPs in some contexts. For more details see [BPSV09].

5.2.4 Simply Prototype Considerations

`Simply` is a tool (the first tool) for easy CSP modelling and solving, whose main novelty is the generation of SMT problem instances in the standard SMT-LIB format as output. Our aim is to take advantage from the improvements that take place from year to year in SMT technology and methods, in order to solve CSPs.

[1] SMT-COMP: The SAT Modulo Theories Competition (`http://www.smtcomp.org`).

	Simply + SMT solver						Other tools				
	Simply	Z3.2	MathSAT-4.2	CVC3-1.5	Barcelogic 1.3	Yices 1.0.10	G12 MINIZINC 0.9	mzn2fzn + ECLiPSe 6.0	mzn2fzn + SICStus 4.0.7	sPEC2SAT 1.1	Comet 1.2
Queens_50	0.22	t.o.	53.00	m.o.	11.72	29.47	**0.22**	2.04	6.98	248.01	t.o.
Queens_100	0.72	t.o.	t.o.	m.o.	389.04	19.22	**0.84**	t.o.	28.51	t.o.	t.o.
Queens_150	1.54	t.o.	t.o.	m.o.	995.94	t.o.	**150.40**	t.o.	256.18	t.o.	t.o.
Bacp_12_6	0.17	0.55	2.53	t.o.	56.98	**0.19**	0.84	t.o.	3.8	m.o.	268.56
Bacp_12_7	0.18	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	m.o.	t.o.
Bacp_12_8	0.22	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	t.o.	m.o.	t.o.
Bacp_12_9	0.21	**0.27**	10.86	t.o.	314.91	0.64	0.94	t.o.	5.3	m.o.	0.51
Bacp_12_10	0.24	**0.24**	14.97	t.o.	190.10	0.79	1.44	t.o.	6.02	m.o.	0.60
Bacp_12_11	0.24	**0.27**	13.60	t.o.	237.50	1.24	1.70	t.o.	7.97	m.o.	19.56
Bacp_12_12	0.26	**0.48**	13.24	t.o.	338.46	1.32	40.59	t.o.	11.32	m.o.	t.o.
Schurl_12_3	0.06	**0.01**	0.08	7.91	0.03	0.02	t.o.	t.o.	0.24	0.38	0.39
Schurl_13_3	0.08	**0.04**	0.07	14.30	0.05	0.05	t.o.	t.o.	0.28	0.55	0.40
Schurl_14_3	0.09	0.23	0.50	18.24	**0.12**	0.16	t.o.	t.o.	0.32	0.50	0.40
Schurl_15_3	0.10	0.35	0.79	29.15	**0.15**	0.18	t.o.	t.o.	0.37	0.73	0.40
Jobshop_54	0.31	**0.12**	0.27	104.97	7.79	2.69	34.13	1.54	33.30	80.41	1.79
Jobshop_55	0.32	**0.20**	0.35	211.48	11.57	3.63	122.16	2.16	t.o.	80.03	11.65
Jobshop_56	0.30	**0.12**	0.46	358.53	12.08	4.37	396.03	3.13	t.o.	88.11	100.01
Jobshop_57	0.30	**0.34**	0.89	475.55	16.05	6.62	1115.09	1.13	t.o.	85.66	892.54
Jobshop_58	0.34	0.10	0.25	134.71	20.75	11.48	**0.09**	1.22	236.64	95.11	0.82

Table 5.1: Benchmarks in Simply.

In this tool we have made a special emphasis on the simplicity of the language and on getting an easy translation to SMT. But there is some room for improvement:

- **Encoding efficiency**. The efficiency of the encoding could be improved either by obtaining less naive translations of constraints or by introducing new theories and logics. For instance, (unidimensional) arrays of integers in `Simply` programs can be flattened into integer variables (as we currently do) or they can be directly translated into SMT array variables or uninterpreted functions. Nevertheless, since SMT solvers highly differ in the treatment given to different theories and logics, more experimentation has to be done in order to decide a suitable encoding for every construct. From the aforementioned experimentation, we would like `Simply` to be able to automatically determine a suitable logic for each problem. The compiler can also be improved in order to obtain SMT formulas without unevaluated subexpressions that could be evaluated at compilation time. For instance `(+ 3 (+ a 10))` should be `(+ a 13)`.

- **`Simply` language richness**. The input language can be extended in several directions: more control structures (*Exists*), more global constraints (At_Most, At_Least, ...), etc.

- **Comparison with others solvers**. The efficiency of the system has been compared with other solvers, but, this comparison has been made using different models for each type of solver. Therefore, we do not know if the differences are due to the efficiency of SMT or to the modelling. It would be convenient to use the same modelling in our tool and in other solvers.

To compare our approach using SMT with other approaches in a more complete and fair (same encoding) mode, we must implement a system that works over a standard language for CSPs. We have used the standard language for specifying CSPs called MiniZinc [Nic07].

5.3 MiniZinc and FlatZinc

MiniZinc, developed as part of the G12 project, aims to be a standard language for specifying CSPs (with or without optimization) over Boolean, integers and floats numbers. Arrays (one and multi dimensional) and sets of integers, floats or Booleans are also supported. MiniZinc is a mostly declarative constraint modelling language, although it also provides some facilities such as *annotations* for specifying, e.g., search strategies, that can be taken into account by the underlying solver. MiniZinc also supports user-defined predicates, some overloading, and some automatic coercions.

Example 21 *This is a MiniZinc toy instance of the well-known Job-shop problem. A Job-shop has some machines, each performing a different operation. There are some jobs to be performed and a job is a sequence of tasks. Each task involves processing by a single machine for some duration and a machine can operate on at most one task at a time. Tasks cannot be interrupted. The goal is, given a deadline, to schedule each job such that its ending time does not exceed the deadline. We continue using this example later on to illustrate the translation process of our tool.*

```
size = 2; d = [| 2,5 | 3,4 |];
int: size; % size of problem
array [1..size,1..size] of int: d; % task durations
int: total = sum(i,j in 1..size) (d[i,j]); % total duration
array [1..size,1..size] of var 0..total: s; % start times
var 0..total: end; % total end time

predicate no_overlap(var int:s1, int:d1, var int:s2, int:d2) =
    s1 + d1 <= s2 \/ s2 + d2 <= s1;

constraint
  forall(i in 1..size) (
    forall(j in 1..size-1) (s[i,j] + d[i,j] <= s[i,j+1]) /\
    s[i,size] + d[i,size] <= end /\
    forall(j,k in 1..size where j < k) (
      no_overlap(s[j,i], d[j,i], s[k,i], d[k,i])
  )
);
solve minimize end;
```

One of the most appealing features of the language is that the specified problems can be easily mapped onto different existing solvers, by previously compiling its model and data files into FLATZINC instances. FLATZINC is a low-level solver input language, for which there exist front-ends for several solvers, such as Gecode [SLT10], ECL^{i}PS^e [AW07], SICStus Prolog [Sic10], JaCoP [Jac10] and SCIP [Sci10], apart from a number of solvers developed by the G12 research team. FLATZINC supports Boolean, integers and floats numbers, one dimensional arrays and sets of all basic types.

FLATZINC instances are simply a list of variable declarations and flat constraints, plus (possibly) a variable to optimize. The MINIZINC to FLATZINC translation has two parts (see [Nic07] for details):

- *Flattening*, where several reductions (built-ins evaluation, list comprehension unrolling, fixed array accesses replacement, etc.) are applied until a fix-point is reached.

- *Post-flattening*, where some normalization is done. For example, for Boolean expressions that are not top-level conjunctions, each sub-expression is replaced by a new Boolean variable, and constraints equating these new variables with the sub-expressions they replaced are added. Similarly is done for non-linear numeric expressions.

Example 22 *This is the FLATZINC instance resulting from translating the MINIZINC instance of Example 21.*

```
var bool: BOOL____1;
var bool: BOOL____2;
var bool: BOOL____3 = true;
var bool: BOOL____4;
var bool: BOOL____5;
var bool: BOOL____6 = true;
array [1..4] of int: d = [2, 5, 3, 4];
var 5..14: end;
array [1..4] of var 0..14: s;
constraint array_bool_or([BOOL____1, BOOL____2], BOOL____3);
constraint array_bool_or([BOOL____4, BOOL____5], BOOL____6);
constraint int_lin_le([-1, 1], [end, s[2]], -5);
constraint int_lin_le([-1, 1], [end, s[4]], -4);
constraint int_lin_le([1, -1], [s[1], s[2]], -2);
constraint int_lin_le([1, -1], [s[3], s[4]], -3);
constraint int_lin_le_reif([1, -1], [s[1], s[3]], -2, BOOL____1);
constraint int_lin_le_reif([-1, 1], [s[1], s[3]], -3, BOOL____2);
constraint int_lin_le_reif([1, -1], [s[2], s[4]], -5, BOOL____4);
constraint int_lin_le_reif([-1, 1], [s[2], s[4]], -4, BOOL____5);
solve minimize end;
```

Note that the arithmetic expressions have been encoded as linear constraints with the `int_lin_le` *constraint.*

The G12 MiniZinc distribution [G1210] is accompanied with a comprehensive set of benchmarks. Moreover, in the MiniZinc challenge [SBF10], which is run every year since 2008 with the aim of comparing different solving technologies on common benchmarks, all entrants are encouraged to submit two models each with a suite of instances to be considered for inclusion in the challenge.

5.4 fzn2smt

To really see that SMT is a good approach to deal with CSPs and COPs we decided to develop an SMT-based FlatZinc solver. This tool should have the following characteristics:

- **Complete FlatZinc support**. To allow comparisons with other solvers (finite domain, MILP, ...) it should support all FlatZinc specifications:
  - Data types: integers, Booleans and floats.
  - Data structures: One dimensional arrays and sets of all basic types.
  - Constraints: constraints over integers, Booleans, floats, arrays and sets.
  - Solve items: satisfaction or minimization/maximization of an integer variable.

- **High performance**. We want to get the best possible encodings. We will use the most appropriate theories, include redundancy and use the type of clauses that give us the best performance.
- **Automatic determination of the theory**. It should be able to determine an appropriate theory for each problem.
- **Choosable optimization approach**. It should support several bounding strategies: linear, dichotomic and hybrid.
- **Possibility of using different SMT solvers**. It should be able to use different SMT solvers such as Yices, Z3, ... with or without APIs.

Our tool is diagrammatically represented in Fig. 5.8, through the process of compiling and solving FLATZINC instances. Shaded boxes (connected by dashed lines) denote inputs and outputs, rounded corner boxes denote actions and diamond boxes denote conditions.

The input of the compiler is a FLATZINC instance which we assume to come from the translation of a MINIZINC one. Hence we are dealing with "safe" FLATZINC instances, e.g., we don't care about out of bounds array accesses. We are also assuming that all global constraints have been reformulated into FLATZINC constraints with the default encoding provided by the MINIZINC distribution.

The input FLATZINC instance is translated into an SMT one (in the standard SMT-LIB format v1.2) and fed into an SMT solver. As a by-product, `fzn2smt` generates the corresponding SMT instance as an output file. Due to the large number of existing SMT solvers, each one supporting different combinations of theories, the user can choose which solver to use (default currently being Yices 2 Prototype[2]).

The FLATZINC language has three solving options, namely: `solve satisfy`, `solve minimize` *obj* and `solve maximize` *obj*, where *obj* is either the name of a variable v or a subscripted array variable $v[i]$, where i is an integer literal. Since optimization is supported neither in the SMT-LIB language nor by most SMT solvers, we have naively implemented it by means of iterative calls successively restricting the domain of the variable to be optimized (as explained in detail in Subsection 5.4.4). Notice from the diagram of Fig. 5.8 that when, after restricting the domain, the SMT solver finds that the problem is not satisfiable anymore, the last previously saved (and hence optimal) solution is recovered. Moreover, since there is no standard output format currently supported by SMT solvers[3], we need a specialized *recovery module* for each solver in order to translate its output to the FLATZINC output format. Currently, `fzn2smt` can recover the output

[2] `http://yices.csl.sri.com/download-yices2.shtml`

[3] There are even solvers that only return `sat`, `unsat` or `unknown`. A proposal of a standard format for solutions has been recently proposed in the SMT-LIB Standard v2.0 [BST10b].

from Yices [DdM06b], Barcelogic [BNO+08b], Z3 [dMB08] and MathSat [BCF+08] SMT solvers.

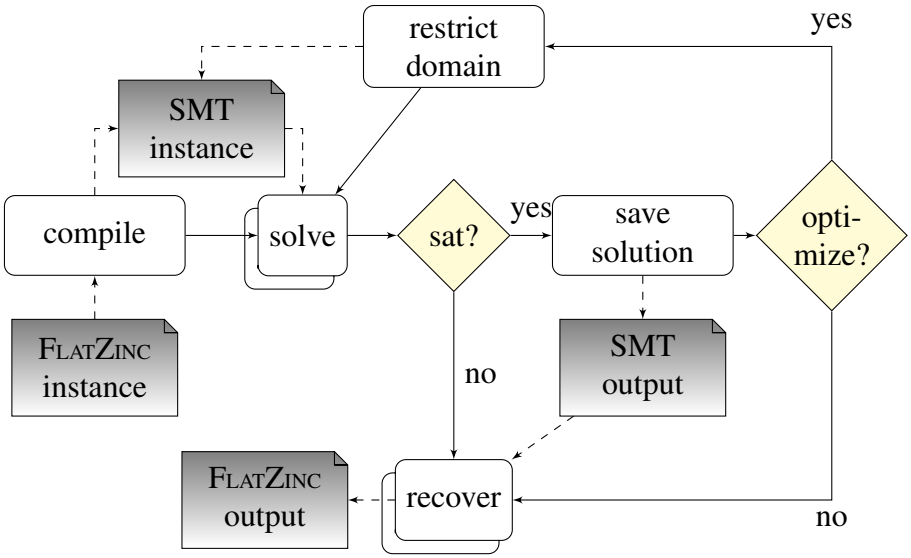


Figure 5.8: The compiling and solving process of fzn2smt.

5.4.1 Translation and Encoding

Since we are encoding FlatZinc instances into SMT, we have to keep in mind two important considerations: on the one hand we have a much richer language than plain SAT, thanks to the theories, and this will allow for more direct translations. On the other hand, in order to take advantage of the SMT solving mechanisms, the more logical structure the SMT formula has, the better. In particular, it is better to introduce clauses instead of expressing disjunctions arithmetically.

A FlatZinc file consists of

1. a list of constant and (finite domain) variable declarations,
2. a list of flat constraints, and
3. a solve goal.

Here we describe the translation of these three basic ingredients.

5.4.2 Constant and Variable Declarations

FLATZINC has two categories of data: constants (also called parameters) and variables (typically with an associated finite domain). Data can be of any of three scalar types: Booleans, integers and floats, or of any of two compound types: sets of integers and one-dimensional $(1..n)$-indexed arrays (multi-dimensional arrays are flattened to arrays of one dimension in the translation from MINIZINC to FLATZINC). Scalar type domains are usually specified by a range or a list of possible values. Our translation of FLATZINC constants and variables is as follows:

- Scalar type constant names are always replaced by their corresponding value.
- Scalar type variable declarations are translated into their equivalent variable declaration in SMT, plus some constraints on the possible values of the SMT variable in order to fix the domain. For example, `var 0..14: x` is translated into the SMT declaration of the integer variable x plus the constraints $x \geq 0$, $x \leq 14$, whereas `var {1,3,7}: x` is translated into the SMT declaration of the integer variable x plus the constraint $x = 1 \vee x = 3 \vee x = 7$.

  Although SMT solvers are able to deal with arbitrarily large integers (as well as with arbitrary precision real numbers), for unrestricted domain integer variables we assume the G12 FLATZINC default domain range of `-10000000..10000000`, i.e., we add the constraints $x \geq -10000000$, $x \leq 10000000$ for every unrestricted integer variable x. This way, the results obtained by our system are consistent with the ones obtained by other tools.
- The domain of a FLATZINC set of integers is specified either by a range or by a list of integers. For this reason, we simply use an occurrence representation by introducing a Boolean variable for every possible element, which indicates whether the element belongs to the set or not. This allows for a simple translation into SMT of most of the constraints involving sets (see Subsection 5.4.3).

  However, in order to be able to translate the set cardinality constraint, which implies counting the number of elements in a set, a $0/1$ partner variable is needed for each introduced Boolean variable. For example, given the declaration `var set of {2,5,6}:s`, we introduce three Boolean variables s_2, s_5 and s_6, three corresponding integer variables s_{i_2}, s_{i_5} and s_{i_6}, the constraints restricting the domain of the integer variables

$$0 \leq s_{i_2},\ s_{i_2} \leq 1$$
$$0 \leq s_{i_5},\ s_{i_5} \leq 1$$
$$0 \leq s_{i_6},\ s_{i_6} \leq 1$$

and the constraints linking the Boolean variables with their integer counterpart

$$s_2 \rightarrow s_{i_2} = 1,\ \neg s_2 \rightarrow s_{i_2} = 0$$
$$s_5 \rightarrow s_{i_5} = 1,\ \neg s_5 \rightarrow s_{i_5} = 0$$
$$s_6 \rightarrow s_{i_6} = 1,\ \neg s_6 \rightarrow s_{i_6} = 0.$$

Hence, the number of SMT variables increases linearly with the size of the domain of the set. Note also that all introduced clauses are either unary or binary, and hence facilitate Boolean unit propagation.

For the case of constant sets no variables are introduced at all. Instead, the element values of the constant set are directly used in the operations involving it, in order to obtain a simpler encoding.

- For the translation of arrays, we provide two options (which can be chosen by a command line option[4]):

  - Using uninterpreted functions: each array is translated into an uninterpreted function of the same name. For example, `array[1..3] of var 1..5:a` is translated into $a : int \mapsto int$. The domain of the elements of the array is constrained as in the scalar case, that is, $1 \leq a(1)$, $a(1) \leq 5$, $1 \leq a(2)$, $a(2) \leq 5$, $1 \leq a(3)$, $a(3) \leq 5$.
  - Decomposing the array into as many base type variables as array elements. For example, the previous array `a` would be decomposed into three integer variables a_1, a_2 and a_3, with the domain constrained analogously to before.

  In the case of constant arrays, equality is used (instead of two inequalities restricting the domain) to state the value of each element. If, moreover, an access to a constant array uses a constant index, we can simply replace that access with its corresponding value. And, if this is the case for all the accesses to an array, then there is no need for introducing an uninterpreted function or a set of base type variables to represent the array.

  Regarding the two possible encodings, the use of uninterpreted functions seems to be more natural, and allows for more compact encodings of array constraints. For example, to express that some element of the previous array a is equal to 1, we simply write

$$1 \leq i,\ i \leq 3$$
$$a(i) = 1$$

[4] Due to our encoding of the operations on sets, arrays of sets are always decomposed into a number of sets.

where i is an integer variable, whereas in the decomposition approach the same statement should be expressed as

$$\begin{aligned} 1 \leq i,\, i \leq 3 \\ i = 1 \rightarrow a_1 = 1 \\ i = 2 \rightarrow a_2 = 1 \\ i = 3 \rightarrow a_3 = 1. \end{aligned}$$

On the other hand, we have ruled out using the SMT theory of arrays. This theory involves $read$ and $write$ operations and, hence, is intended to be used for modelling state change of imperative programs with arrays. But, since it makes no sense thinking of $write$ operations on arrays in the setting of CP, it suffices to translate every expression of the form $read(a, i)$ into $a(i)$, where a is an uninterpreted function. Moreover, deciding satisfiability of sets of atomic constraints involving uninterpreted functions is far cheaper than using the arrays theory.

However, the uninterpreted functions approach still has the drawback of using more than one theory, namely, uninterpreted functions (UF) and linear integer arithmetic (LIA), and suffers from a non-negligible computational overhead due to theory combination. In Section 5.2.3 a performance comparison of the two possible encodings for arrays is given.

5.4.3 Constraints

The second and main block of a FlatZinc file is the set of constraints that a solution must satisfy. The arguments of these flat constraints can be literal values, constant or variable names, or subscripted array constants or variables $v[i]$ where i in an integer literal. A literal value can be either a value of scalar type, an explicit set (e.g., {`2,3,5`}) or an explicit array `[`$y_1, \ldots, y_k$`]`, where each array element y_i is either a non-array literal, the name of a non-array constant or variable, or a subscripted array constant or variable $v[i]$, where i in an integer literal (e.g., `[x,a[1],3]`).

We perform the following translation of constraint arguments prior to translating the constraints. Constant names are replaced by their value. Scalar type variables are replaced by their SMT counterpart. Finally, array accesses are translated depending on the chosen array treatment (see the previous subsection): when using uninterpreted functions, an array access $v[i]$ is translated into $v(i)$, where v is an uninterpreted function; when decomposing the array into base type variables, $v[i]$ is translated into v_i (the corresponding variable for that position of the array). We remark that, in all array accesses $v[i]$, i can be assumed to be an integer literal, i.e., i cannot be a variable, since all variable subscripted array expressions are replaced by `array_element` constraints during the translation

from MINIZINC to FLATZINC. Moreover, we don't need to perform array bounds checks, because this is already done by the MINIZINC to FLATZINC compiler.

In the following we describe the translation of the constraints, that we have categorized into *Boolean constraints*, *Integer constraints*, *Float constraints*, *Array constraints* and *Set constraints*.

- *Boolean constraints* are built with the common binary Boolean operators (and, or, $implies$, ...) and the relational operators ($<$, $\leq$, $=$, ...) over Booleans. All of them have their counterpart in the SMT-LIB language, and hence have a direct translation.

  There is also the `bool2int(a,n)` constraint, which maps a Boolean variable into a 0/1 integer. We translate it into $(a \rightarrow n = 1) \wedge (\neg a \rightarrow n = 0)$.

- *Integer constraints* are built with the common relational constraints over integer expressions (hence they are straightforwardly translated into SMT). They also include some named constraints. Here we give the translation of some representative ones.

  The constraint

  $$\texttt{int\_lin\_eq}([\texttt{c}_1, \ldots, \texttt{c}_\texttt{n}], [\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}], \texttt{r})$$

  where $\texttt{c}_1, \ldots, \texttt{c}_\texttt{n}$ are integer constants and $\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}$ are integer variables or constants, means, and is translated as

  $$\sum_{i \in 1..n} c_i v_i = r.$$

  The minimum constraint `int_min(a,b,c)`, meaning $min(a, b) = c$, is translated as

  $$(a > b \rightarrow c = b) \wedge (a \leq b \rightarrow c = a).$$

  The absolute value constraint `int_abs(a,b)`, meaning $|a| = b$, is translated as

  $$(a = b \vee -a = b) \wedge b \geq 0.$$

  The constraint `int_times(a,b,c)`, that states $a \cdot b = c$, can be translated into a set of linear arithmetic constraints under certain circumstances: if either a or b are (uninstantiated) finite domain variables, we linearize this constraint by conditionally instantiating the variable with the smallest domain, e.g.,

  $$\bigwedge_{i \in \mathcal{D}om(a)} (i = a \rightarrow i \cdot b = c).$$

In fact, we do it better (i.e., we do not necessarily expand the expression for all the values of $\mathcal{D}om(a)$) by considering the domain bounds of b and c, and narrowing accordingly the domain of a.

Since it is better to use the simplest logic at hand, we use linear integer arithmetic for the translation if possible, and non-linear integer arithmetic otherwise. Hence, only in the case that a and b are unrestricted domain variables we translate the previous constraint as $a \cdot b = c$ and label the SMT instance to require QF_NIA (*non-linear integer arithmetic logic*).

- *Float constraints* are essentially the same as the integer ones, but involving float data. Hence, the translation goes in the same way as for the integers, except that the inferred logic is QF_LRA (*linear real arithmetic*).

- *Array constraints*. The main constraint dealing with arrays is `element`, which restricts an element of the array to be equal to some variable. As an example, `array_var_int_element(i, a, e)` states $i \in 1..n \wedge a[i] = e$, where n is the size of a. The translation varies depending on the representation chosen for arrays (see the previous subsection):

  - In the uninterpreted functions approach, the translation is

    $$1 \leq i \wedge i \leq n \wedge a(i) = e,$$

    where a is the uninterpreted function symbol representing the array `a`.
  - In the decomposition approach, the translation is

    $$1 \leq i \wedge i \leq n \wedge \left(\bigwedge_{j \in 1..n} i = j \rightarrow a_j = e \right).$$

  Constraints such as `array_bool_and`, `array_bool_or` or `bool_clause`, dealing with arrays of Boolean, are straightforwardly translated into SMT.

- *Set constraints*. These are the usual constraints over sets. We give the translation of some of them.

  The constraint `set_card(s,k)`, stating $|s| = k$, is translated by using the $0/1$ variables introduced for each element (see previous subsection) as $\sum_{j \in \mathcal{D}om(s)} s_{i_j} = k$.

  The constraint `set_in(e,s)`, stating $e \in s$, is translated depending on whether e is instantiated or not. If e is instantiated then `set_in(e,s)` is translated as s_e if e is in the domain of s (recall that we are introducing a Boolean variable s_e for each

element e in the domain of s), and as *false* otherwise. If `e` is not instantiated, then we translate the constraint as

$$\bigvee_{j \in \mathcal{D}om(s)} (e = j) \wedge s_j.$$

For constraints involving more than one set, one of the difficulties in their translation is that the involved sets can have distinct domains. For example, the constraint `set_eq(a,b)`, stating $a = b$, is translated as

$$\left(\bigwedge_{j \in \mathcal{D}om(a) \cap \mathcal{D}om(b)} a_j = b_j \right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(a) \setminus \mathcal{D}om(b)} \neg a_j \right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(b) \setminus \mathcal{D}om(a)} \neg b_j \right).$$

And the constraint `set_diff(a,b,c)`, which states $a \setminus b = c$, is translated as

$$\left(\bigwedge_{j \in (\mathcal{D}om(a) \setminus \mathcal{D}om(b)) \cap \mathcal{D}om(c)} a_j = c_j \right) \wedge \left(\bigwedge_{j \in (\mathcal{D}om(a) \setminus \mathcal{D}om(b)) \setminus \mathcal{D}om(c)} \neg a_j \right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(c) \setminus \mathcal{D}om(a)} \neg c_j \right) \wedge \left(\bigwedge_{j \in \mathcal{D}om(a) \cap \mathcal{D}om(b) \cap \mathcal{D}om(c)} \neg c_j \right) \wedge \left(\bigwedge_{j \in (\mathcal{D}om(a) \cap \mathcal{D}om(b)) \setminus \mathcal{D}om(c)} a_j \rightarrow b_j \right).$$

Although the translation of the set constraints seem to be convoluted, note that we are mainly introducing unit and binary clauses. We remark that when the sets are already instantiated at compilation time, some simplifications are actually made. Note also that the size of the SMT formula increases linearly on the size of the domains of the sets.

Finally, let us mention that almost all FLATZINC constraints have a reified counterpart. For example, in addition to the constraint `int_le(a,b)`, stating $a \leq b$, there is a constraint `int_le_reif(a,b,r)`, stating $a \leq b \leftrightarrow r$, where a and b are integer variables and r is a Boolean variable. In all these cases, given the translation of a constraint, the translation of its reified version into SMT is direct.

5.4.4 Solve Goal

A FlatZinc file must end with a solve goal, which can be of one of the following forms: `solve satisfy`, for checking satisfiability and providing a solution if possible, or `solve minimize` obj or `solve maximize` obj, for looking for a solution that minimizes or maximizes, respectively, the value of obj, where obj is either a variable v or a subscripted array variable $v[i]$, where i is an integer literal. Although search annotations can be used in the solve goal, they are currently ignored in our tool.

When the `satisfy` option is used, we just need to feed the selected SMT solver with the SMT file resulting from the translation explained in the previous subsections (see Example 23 to see a complete SMT-LIB instance generated by `fzn2smt`). Thereafter the recovery module will translate the output of the SMT solver to the FlatZinc format.

Since most SMT solvers do not provide optimization facilities[5], we have implemented an ad hoc search procedure in order to deal with the `minimize` and `maximize` options. This procedure successively calls the SMT solver with different problem instances, by restricting the domain of the variable to be optimized with the addition of constraints. We have implemented three possible bounding strategies: linear, dichotomic and hybrid. The linear bounding strategy approaches the optimum from the satisfiable side[6], while the dichotomic strategy simply consists of binary search optimization. Finally, the hybrid strategy makes a preliminary approach to the optimum by means of binary search and, when a (user definable) threshold on the possible domain of the variable is reached, it turns into the linear approach, again from the satisfiable side. Both the bounding strategy and the threshold for the hybrid case can be specified by the user from the command line. Algorithm 14 describes our optimization procedure, taking into account all the bounding strategies, for the minimization case.

The `SMT_solve` function consists of a call to the SMT solver with an SMT instance. The `bound(`SMT_inst`,` var`,` inf`,` sup`)` function returns the SMT instance resulting from adding the bounds $var \geq inf$ and $var \leq sup$ to the SMT instance SMT_inst.

In the current implementation we are not keeping learnt clauses from one iteration of the SMT solver to the next since we are using the SMT solver as a black box. We have also tested an implementation of these optimization algorithms using the Yices API, which allows keeping the learnt clauses, without obtaining significantly better results.

[5]There are however some solvers, such as Yices and Z3, that already provide MaxSMT facilities. On the other hand, the problem of optimization modulo theories has been recently addressed in [CFG+10], by introducing a theory of costs.

[6]This allows us to eventually jump several values when we find a new solution. On the contrary, approaching from the unsatisfiable side is only possible by modifying the value of the variable to optimize in one unit at each step.

Algorithm 14 Minimization in fzn2smt

Input: $SMT\_inst : SMTinstance$;
$var : int$; // variable to minimize
$inf, sup : int$; // bounds of the variable to minimize
$t : int$; // threshold for the hybrid strategy
$bs : \{linear, binary, hybrid\}$; // bounding strategy
Output: $\langle int,\ sat|unsat \rangle$
$\langle sol, status \rangle$:= SMT_solve(bound(SMT_inst, var, inf, sup));
if $status$ = unsat **then**
  return $\langle \infty, unsat \rangle$;
else
  $sup := sol - 1$;
  while $inf \leq sup$ **do**
    if $(sup - inf \leq t \ \wedge \ bs = hybrid)$ **then**
      bs := linear;
    end if
    if bs = linear **then**
      $med := sup$;
    else
      $med := \lceil (inf + sup)/2 \rceil$;
    end if
    $\langle new\_sol, status \rangle$:= SMT_solve(bound(SMT_inst, var, inf, med));
    if $status$ = unsat **then**
      if bs = linear **then**
        return $\langle sol, sat \rangle$;
      else
        $inf := med + 1$;
      end if
    else
      $sol := new\_sol$;
      $sup := sol - 1$;
    end if
  end while
  return $\langle sol, sat \rangle$;
end if

Example 23 *Continuing Example 22, here follows the SMT-LIB instance produced by* `fzn2smt`*. For the minimization process, we should add the appropriate bounds to the objective variable* `end`*.*

```
(benchmark jobshop.fzn.smt
  :source { Generated by fzn2smt }
  :logic QF_IDL
  :extrapreds ((BOOL____4) (BOOL____2) (BOOL____1) (BOOL____5))
  :extrafuns ((s_1_ Int) (s_2_ Int) (s_3_ Int) (s_4_ Int) (end Int))
  :formula (and
    (>= end 5)
    (<= end 14)
    (>= s_1_ 0)
    (<= s_1_ 14)
    (>= s_2_ 0)
    (<= s_2_ 14)
    (>= s_3_ 0)
    (<= s_3_ 14)
    (>= s_4_ 0)
    (<= s_4_ 14)
    (= (or  BOOL____1 BOOL____2) true)
    (= (or  BOOL____4 BOOL____5) true)
    (<= (+ (~ end) s_2_) (~ 5))
    (<= (+ (~ end) s_4_) (~ 4))
    (<= (+ s_1_ (~ s_2_)) (~ 2))
    (<= (+ s_3_ (~ s_4_)) (~ 3))
    (= (<= (+ s_1_ (~ s_3_)) (~ 2)) BOOL____1)
    (= (<= (+ (~ s_1_) s_3_) (~ 3)) BOOL____2)
    (= (<= (+ s_2_ (~ s_4_)) (~ 5)) BOOL____4)
    (= (<= (+ (~ s_2_) s_4_) (~ 4)) BOOL____5)
  )
)
```

5.5 Benchmarking

In this section we compare the performance of `fzn2smt` and that of several existing FlatZinc solvers on FlatZinc instances, and provide some possible explanations about the `fzn2smt` behavior. We first compare several SMT solvers within `fzn2smt` and, then, use the one with the best results to compare against other existing FlatZinc solvers.

We perform the comparisons on the benchmarks of the three MiniZinc challenge competitions (2008, 2009 and 2010), consisting of a total of 294 instances from 32 problems. These benchmarks consist of a mixture of puzzles, planning, scheduling and graph problems. Half of the problems are optimization problems, whilst the other half are satisfiability ones.

We present several tables that, for each solver and problem, report the accumulated time for the solved instances and the number of solved instances (within parenthesis). The times are the sum of the translation time, when needed (e.g., `fzn2smt` translates from FlatZinc to the SMT-LIB format), plus the solving time. We indicate in boldface the cases with more solved instances, breaking ties by total time. The experiments have been executed on an Intel® Core™ i5 CPU at 2.66 GHz, with 6GB of RAM, running 64-bit openSUSE 11.2 (kernel 2.6.31), with a time limit of 15 minutes per instance (the same as in the competition).

5.5.1 fzn2smt with SMT Solvers

Here we compare the performance of several SMT solvers which are SMT-LIB 1.2 compliant, working in cooperation with `fzn2smt` v2.0.1, on the MiniZinc challenge benchmarks. We have selected the solvers that historically have had good performance in the QF_LIA division of the annual SMT competition. These are Barcelogic 1.3 [BNO+08b], MathSAT 5 (successor of MathSAT 4 [BCF+08]; still work in progress), Yices 1.0.28 [DdM06b], Yices 2 Prototype (still work in progress), and Z3.2 [dMB08]. Linux binaries of most of these solvers can be downloaded from `http://www.smtcomp.org`.

In the executions of this subsection we have used the default options for `fzn2smt`: array expansion (see Subsection and binary search for optimization (see Subsection 5.4.4).

In Table 5.2 we can observe that Yices 1.0.28 and Yices 2 are able to solve, respectively, 213 and 212 (out of 294) instances. We consider performance of Yices 2 the best of those considered because it had the best performance on 19 of the problems, far more than any of the other solvers.

5.5.2 Array Encodings

In Table 5.3 we compare the performance of using array decomposition versus uninterpreted functions as array encodings. We only give the results for Yices 2 proto, which is the solver with the best performance in the executions of Table 5.2 (where array decomposition was used as default option).

As shown in Table 5.3, the decomposition approach clearly outperforms the uninterpreted functions approach on FlatZinc instances from the MiniZinc distribution. We have also tested other SMT solvers than Yices, obtaining similar results. This apparently strange behaviour is better understood when looking at how SMT solvers deal with uninterpreted functions and, in particular, how this behaves on the instances generated by our tool. Hence, first of all, let us see some possible treatments to uninterpreted functions in the context of SMT. The reader can refer to [BCF+06] for a deeper discussion on this

Table 5.2: Comparison of SMT solver using fzn2smt.

Problem	Type	#	Barcelogic 1.3		MathSAT 5		Yices 1.0.28		Yices 2 proto		Z3.2	
debruijn-binary	s	11	0.60	(1)	0.56	(1)	**0.47**	**(1)**	0.50	(1)	0.57	(1)
nmseq	s	10	0.00	(0)	568.35	(2)	778. 07	(5)	118.74	(2)	**173.84**	**(5)**
pentominoes	s	7	0.00	(0)	291.60	(1)	**50.47**	**(2)**	168.47	(2)	314.48	(1)
quasigroup7	s	10	34.16	(2)	191.83	(5)	54.94	(5)	**20.03**	**(5)**	691.32	(4)
radiation	o	9	342.34	(1)	2202.92	(9)	**373.24**	**(9)**	1047.03	(9)	2473.27	(9)
rcpsp	o	10	0.00	(0)	315.46	(2)	920.08	(8)	**993.74**	**(9)**	1791.79	(8)
search-stress	s	3	0.84	(2)	1.05	(2)	0.86	(2)	**0.74**	**(2)**	0.84	(2)
shortest-path	o	10	**109.35**	**(9)**	484.73	(8)	658.58	(9)	1419.67	(7)	790.54	(8)
slow-convergence	s	10	405.25	(7)	426.79	(7)	269.66	(7)	**247.06**	**(7)**	291.99	(7)
trucking	o	10	254.11	(5)	31.70	(4)	**9. 50**	**(5)**	47.77	(5)	1084.97	(4)
black-hole	s	10	511.13	(1)	29.24	(1)	**3857.51**	**(9)**	892.46	(8)	765.09	(1)
fillomino	s	10	93.87	(10)	30.28	(10)	20.48	(10)	**19.99**	**(10)**	21.13	(10)
nonogram	s	10	0.00	(0)	0.00	(0)	**1656.54**	**(10)**	1546.56	(7)	0.00	(0)
open-stacks	o	10	1772.39	(5)	0.00	(0)	702.09	(6)	**707.25**	**(7)**	776.61	(6)
p1f	o	10	875.54	(8)	**86.90**	**(9)**	167.84	(9)	126.01	(9)	184.22	(9)
prop-stress	s	10	315.80	(7)	330.31	(7)	**266.11**	**(7)**	274.07	(7)	289.23	(7)
rect-packing	s	10	559.50	(5)	679.62	(10)	**104.82**	**(10)**	106.66	(10)	122.28	(10)
roster-model	o	10	98.41	(10)	51.03	(10)	53.89	(10)	**50.38**	**(10)**	56.04	(10)
search-stress2	s	10	23.19	(10)	14.63	(10)	9.43	(10)	**7.90**	**(10)**	10.55	(10)
still-life	o	4	30.64	(3)	132.51	(4)	128.71	(4)	**62.18**	**(4)**	173.82	(4)
vrp	o	10	0.00	(0)	0.00	(0)	0.00	(0)	0.00	(0)	0.00	(0)
costas-array	s	5	0.00	(0)	0.00	(0)	675.23	(1)	**664.13**	**(2)**	628.83	(1)
depot-placement	o	15	2480.74	(5)	2496.69	(10)	613.53	(15)	**295.78**	**(15)**	2073.93	(15)
filter	o	10	24.02	(6)	38.01	(6)	24.28	(6)	**17.88**	**(6)**	22.42	(6)
ghoulomb	o	10	0.00	(0)	0.00	(0)	75.87	(1)	**2545.02**	**(6)**	508.19	(2)
gridColoring	o	5	4.82	(2)	202.18	(3)	857.74	(3)	**38.15**	**(3)**	51.94	(3)
rcpsp-max	o	10	85.04	(1)	238.34	(2)	533.87	(4)	475.14	(4)	**383.73**	**(4)**
solbat	s	15	0.00	(0)	1721.73	(15)	341.04	(15)	**141.48**	**(15)**	1339.75	(15)
sugiyama2	o	5	79.38	(5)	21.31	(5)	10.26	(5)	**8.64**	**(5)**	9.56	(5)
wwtp-random	s	5	61.08	(5)	29.72	(5)	17.71	(5)	**15.21**	**(5)**	16.56	(5)
wwtp-real	s	5	107.57	(5)	39.77	(5)	17.97	(5)	**14.47**	**(5)**	17.09	(5)
bacp	o	15	1753.26	(14)	232.22	(15)	75.16	(15)	**64.17**	**(15)**	97.51	(15)
Total		294	10023	(129)	10889	(168)	**13325**	**(213)**	12137	(212)	15162	(192)

Performance of state-of-the-art SMT solvers in cooperation with `fzn2smt` on the MiniZinc challenge benchmarks. Type 's' stands for satisfaction and 'o' for optimization. # stands for the number of instances.

Table 5.3: Performance with Yices 2 using array decomposition vs uninterpreted functions (UF).

Problem	Type	#	Decomposition		UF	
debruijn-binary	s	11	**0.50**	**(1)**	0.74	(1)
nmseq	s	10	118.74	(2)	**83.51**	**(2)**
pentominoes	s	7	**168.47**	**(2)**	197.22	(1)
quasigroup7	s	10	**20.03**	**(5)**	336.59	(5)
radiation	o	9	**1047.03**	**(9)**	2360.85	(9)
rcpsp	o	10	**993.74**	**(9)**	695.84	(8)
search-stress	s	3	**0.74**	**(2)**	0.84	(2)
shortest-path	o	10	**1419.67**	**(7)**	1068.03	(4)
slow-convergence	s	10	**247.06**	**(7)**	522.42	(3)
trucking	o	10	47.77	(5)	**39.67**	**(5)**
black-hole	s	10	**892.46**	**(8)**	0.00	(0)
fillomino	s	10	19.99	(10)	**19.21**	**(10)**
nonogram	s	10	**1546.56**	**(7)**	0.00	(0)
open-stacks	o	10	**707.25**	**(7)**	1729.37	(5)
p1f	o	10	**126.01**	**(9)**	25.49	(8)
prop-stress	s	10	274.07	(7)	**262.95**	**(7)**
rect-packing	s	10	**106.66**	**(10)**	112.10	(10)
roster-model	o	10	**50.38**	**(10)**	51.01	(10)
search-stress2	s	10	**7.90**	**(10)**	9.74	(10)
still-life	o	4	**62.18**	**(4)**	97.39	(4)
vrp	o	10	0.00	(0)	0.00	(0)
costasArray	s	5	**664.13**	**(2)**	151.60	(1)
depot-placement	o	15	**295.78**	**(15)**	2651.71	(10)
filter	o	10	**17.88**	**(6)**	23.12	(6)
ghoulomb	o	10	**2545.02**	**(6)**	707.74	(2)
gridColoring	o	5	**38.15**	**(3)**	335.10	(3)
rcpsp-max	o	10	**475.14**	**(4)**	498.89	(4)
solbat	s	15	**141.48**	**(15)**	567.69	(15)
sugiyama2	o	5	**8.64**	**(5)**	9.04	(5)
wwtp-random	s	5	**15.21**	**(5)**	34.67	(5)
wwtp-real	s	5	**14.47**	**(5)**	28.82	(5)
bacp	o	15	**64.17**	**(15)**	77.85	(15)
Total		294	**12137**	**(212)**	12699	(175)

issue.

When dealing with two or more theories, a standard approach is to handle the in-

tegration of the different theories by performing some sort of search on the equalities between their shared (or *interface*) variables. First of all, formulas are purified by replacing terms with fresh variables, so that each literal only contains symbols belonging to one theory. For example,

$$a(1) = x + 2$$

is translated into

$$\begin{aligned} a(v_1) &= v_2 \\ v_1 &= 1 \\ v_2 &= x + 2 \end{aligned}$$

where the first literal belongs to UF, and the rest belong to LIA. The variables v_1, v_2 are then called *interface variables*, as they appear in literals belonging to different theories. An *interface equality* is an equality between two interface variables. All theory combination schemata, e.g., Nelson-Oppen [NO79], Shostak [Sho84], or Delayed Theory Combination (DTC) [BBC$^+$06], rely to some point on checking equality between interface variables, in order to ensure mutual consistency between theories. This may imply to assign a truth value to up to all the interface equalities. Since the number of interface equalities is given by $|\mathcal{V}| \cdot (|\mathcal{V}| - 1)/2$, where $|\mathcal{V}|$ is the number of interface variables, the search space may be enlarged in a quadratic factor in the number of interface variables.

In the case of combining UF with another theory T, an alternative approach is to eliminate the uninterpreted function symbols by means of Ackermann's reduction [Ack68], and then solve the resulting SMT problem with only theory T. In Ackermann's reduction, each application $f(a)$ is replaced by a variable f_a, and for each pair of applications $f(a)$, $f(b)$ the formula $a = b \rightarrow f_a = f_b$ is added, i.e., the single theory axiom $x = y \rightarrow f(x) = f(y)$ of the UF theory becomes instantiated as necessary. This is the approach taken by most state-of-the-art SMT solvers. However, this has the same disadvantage as theory combination in that the number of additional literals is quadratic in the size of the input and, in fact, as shown in [BCF$^+$06], there is no clear winner between DTC and Ackermannization.

It is worth noting that current SMT solvers have been designed mainly to deal with verification problems, where there are few parameters and almost all variable values are undefined. In such problems, uninterpreted functions are typically used to abstract pieces of code and, hence, their arguments are variables (or expressions using variables). Moreover, the number of such abstractions is limited. This makes Ackermannization feasible in practice. On the contrary, in the instances we are considering, we have the opposite situation: a lot of parameters and a few decision variables. In particular, most arrays are parameters containing data. For example, in a scheduling problem, a FlatZinc array containing durations of tasks, such as

```
array[1..100] of int:  d = [2,5,...,4];
```

could be expressed using an SMT uninterpreted function as follows:

$$\begin{gathered} d(1) = 2 \\ d(2) = 5 \\ \dots \\ d(100) = 4. \end{gathered}$$

Similarly, for an undefined array containing, e.g., starting times, such as

```
array[1..100] of var 0..3600:  s;
```

we could use an uninterpreted function, and state its domain as follows:

$$\begin{gathered} 0 \leq s(1),\ s(1) \leq 3600 \\ 0 \leq s(2),\ s(2) \leq 3600 \\ \dots \\ 0 \leq s(100),\ s(100) \leq 3600. \end{gathered}$$

In any case, lots of distinct uninterpreted function applications appear, and Ackermannization results in a quadratic number of formulas like $1 = 2 \rightarrow f_1 = f_2$, which are trivially true since the antecedent is false. Although difficult to determine because we are using each SMT solver as a black box, we conjecture that this is not checked in the Ackermannization process since, as said before, uninterpreted functions are expected to have variables in the arguments.

Finally, although the decomposition approach exhibits better performance than the uninterpreted functions approach, we have decided to maintain both options in our system. The main reason is that the uninterpreted functions approach allows for more compact and natural representations and, hopefully, can lead to better results in the future if Ackermannization is adapted accordingly.

5.5.3 Bounding Strategy

Here we test the performance of Yices 2 with the different bounding strategies described in Subsection 5.4.4 for optimization problems. For the hybrid strategy we have used the default threshold of 10 units for switching from the binary to the linear approximation strategy. Experiments with larger threshold have not yielded better results.

Table 5.4 shows that the binary and hybrid strategies perform better than the linear one in general. Both the binary and hybrid strategies are able to solve the same number of instances, but the first one spends less time globally. Nevertheless, the hybrid strategy is

Table 5.4: Performance with Yices 2 using different optimization search strategies.

Problem	#	Binary		Hybrid		Linear	
radiation	9	**1047.03**	**(9)**	1304.59	(9)	2008.77	(9)
rcpsp	10	**993.74**	**(9)**	710.77	(8)	1180.11	(5)
shortest-path	10	1419.67	(7)	1381.92	(7)	**1034.22**	**(8)**
trucking	10	47.77	(5)	41.86	(5)	**34.66**	**(5)**
open-stacks	10	707.25	(7)	**650.27**	**(7)**	691.46	(7)
p1f	10	126.01	(9)	**125.44**	**(9)**	188.14	(9)
roster-model	10	50.38	(10)	50.29	(10)	**50.16**	**(10)**
still-life	4	**62.18**	**(4)**	118.54	(4)	119.39	(4)
vrp	10	0.00	(0)	0.00	(0)	0.00	(0)
depot-placement	15	295.78	(15)	**248.19**	**(15)**	263.61	(15)
filter	10	17.88	(6)	**17.37**	**(6)**	18.34	(6)
ghoulomb	10	**2545.02**	**(6)**	2825.16	(6)	1255. 42	(3)
gridColoring	5	38.15	(3)	**17.43**	**(3)**	17.81	(3)
rcpsp-max	10	475.14	(4)	**460.08**	**(4)**	1035.02	(4)
sugiyama2	5	8.64	(5)	**8.37**	**(5)**	9.25	(5)
bacp	15	64.17	(15)	**58.03**	**(15)**	64.98	(15)
Total	153	**7898**	**(113)**	8018	(113)	7971	(108)

faster than the binary in most of the problems. And, curiously, the linear strategy is better in three problems.

We want to remark that the linear strategy approaches the optimum from the satisfiable side. We also tried the linear strategy approaching from the unsatisfiable side but the results where a bit worse globally. This is probably due to the fact that this last strategy can only make approximation steps of size one whilst the former can make bigger steps when a solution is found. Moreover, the formula resulting from the translation of many MiniZinc benchmarks has very simple Boolean structure (the formula is often trivially satisfiable at the Boolean level), and hence it is likely that the SMT solver cannot substantially profit from conflict-driven lemma learning on unsatisfiable instances. In fact, there exist unsatisfiable instances that result in a few or no conflicts at all, and most of the work is hence done by the theory solver.

5.5.4 Other FlatZinc Solvers

In this section we compare the performance of `fzn2smt` (using Yices 2) and the following available FlatZinc solvers: Gecode (winner of all MiniZinc challenges), G12 and G12 Lazy_fd (the solvers distributed with MiniZinc) and FznTini (a SAT based solver).

Let us remark that `fzn2smt` with Yices 2 obtained (*ex aequo* with Gecode) the

golden medal in the *par* division and the silver medal in the *free* division of the MiniZinc challenge 2010, and the silver medal in the same divisions of the MiniZinc challenge 2011. It is also fair to notice that the solver with the best performance in the MiniZinc challenges 2010 and 2011 (in all categories) was Chuffed, implemented by the MiniZinc team and not eligible for prizes.[7]

Table 5.5 shows the results of this comparison without using solver specific global constraints, which means that global constraints are decomposed into conjunctions of simpler constraints. However, search strategy annotations are enabled in all experiments and, while `fzn2smt` ignores them, the other systems can profit from these annotations.

We can observe that `fzn2smt` is the solver which is able to solve the largest number of instances, closely followed by G12 Lazy_fd and Gecode. Looking at the problems separately, `fzn2smt` offers better performance in 12 cases, followed by G12 Lazy_fd in 10 cases and Gecode in 9 cases.

We remark that G12 Lazy_fd does not support instances with unbounded integer variables: the ones of debruijn-binary, nmseq, open-stacks, p1f, wwtp-random. We have tried to solve these instances by bounding those variables with the MiniZinc standard default limits, i.e., by setting `var -10000000..10000000 : x;` for every unrestricted integer variable x (similarly as we have done for `fzn2smt`), but G12 Lazy_fd runs out of memory. This is probably due to its use of Boolean encodings for the domains of the integer variables, as these encodings imply introducing a new Boolean variable for each element of the domain (see [OSC09]).

The plot of Figure 5.9 shows the elapsed times, in logarithmic scale, for the solved instances of Table 5.5. The instances have been ordered by its execution time in each system. The overall best system (in terms of number of solved instances) is `fzn2smt`. However `fzn2smt` is the worst system (in terms of execution time) within the first 50 solved instances and, moreover, Gecode is better along the 160 first instances, closely followed by G12 Lazy_fd. It must be taken into account that the `fzn2smt` compiler is written in Java, and it generates an SMT file for each decision problem that is fed into the chosen SMT solver. Hence this can cause an overhead in the runtime that can be more sensible for the easier instances. Finally, note also that `fzn2smt` scales very well from the 50 to the 150 first solved instances. This exhibits the SMT solvers robustness.

5.5.5 Other FlatZinc Solvers with Global Constraints

Some FlatZinc solvers provide specific algorithms for certain global constraints (such as `alldifferent`, `cumulative`, etc.). Thus, the user can choose not to decompose some global constraints during the translation from MiniZinc to FlatZinc, in order to profit

[7] See http://www.g12.csse.unimelb.edu.au/minizinc/challenge2011/results2011.html for details.

Table 5.5: Performance comparison between fzn2smt and some available FLATZINC solvers.

n	Problem	Type	#	Gecode		FZNTini		G12		G12 Lazy_fd		fzn2smt	
1	debruijn-binary	s	11	**4.46**	**(6)**	0.06	(1)	31.30	(6)	0.00	(0)	0.50	(1)
2	nmseq	s	10	**535.62**	**(8)**	2.64	(1)	927.42	(7)	0.00	(0)	118.74	(2)
3	pentominoes	s	7	**601.82**	**(7)**	89.68	(1)	848.17	(4)	466.57	(5)	168.47	(2)
4	quasigroup7	s	10	**278.73**	**(6)**	773.28	(4)	1.72	(5)	2.85	(5)	20.30	(5)
5	radiation	o	9	1112.14	(9)	4260.37	(7)	1302.79	(9)	**3.60**	**(9)**	1047.03	(9)
6	rcpsp	o	10	12.07	(5)	0.00	(0)	97.27	(5)	82.60	(8)	**993.74**	**(9)**
7	search-stress	s	3	11.30	(2)	391.71	(3)	14.66	(2)	**0.40**	**(3)**	0.74	(2)
8	shortest-path	o	10	442.49	(10)	0.00	(0)	4.27	(4)	**127.77**	**(10)**	1419.67	(7)
9	slow-convergence	s	10	**8.41**	**(10)**	62.62	(4)	95.42	(10)	154.83	(10)	247.06	(7)
10	trucking	o	10	**1.01**	**(5)**	593.23	(4)	4.12	(5)	159.84	(5)	47.77	(5)
11	black-hole	s	10	69.68	(7)	0.00	(0)	2423.48	(6)	97.69	(7)	**892.46**	**(8)**
12	fillomino	s	10	118.62	(10)	4.36	(10)	332.56	(10)	**2.59**	**(10)**	19.99	(10)
13	nonogram	s	10	1353.63	(8)	48.13	(7)	336.93	(2)	**1533.07**	**(9)**	1546.56	(7)
14	open-stacks	o	10	**169.74**	**(8)**	1325.98	(4)	299.55	(8)	0.00	(0)	707.25	(7)
15	p1f	o	10	2.57	(8)	315.65	(9)	2.40	(8)	0.00	(0)	**126.01**	**(9)**
16	prop-stress	s	10	600.80	(4)	223.52	(2)	883.08	(3)	**221.95**	**(9)**	274.07	(7)
17	rect-packing	s	10	134.36	(6)	569.57	(3)	339.80	(6)	165.58	(6)	**106.66**	**(10)**
18	roster-model	o	10	**1.04**	**(10)**	0.00	(0)	4.46	(10)	18.80	(7)	50.38	(10)
19	search-stress2	s	10	296.16	(9)	9.10	(10)	381.82	(8)	**0.08**	**(10)**	7.90	(10)
20	still-life	o	4	1.01	(3)	35.50	(3)	2.56	(3)	18.55	(3)	**62.18**	**(4)**
21	vrp	o	10	0.00	(0)	0.00	(0)	0.00	(0)	0.00	(0)	0.00	(0)
22	costas-array	s	5	**943.75**	**(4)**	405.28	(2)	423.09	(3)	145.54	(2)	664.13	(2)
23	depot-placement	o	15	1205.75	(12)	1820.57	(8)	522.81	(8)	613.51	(12)	**295.78**	**(15)**
24	filter	o	10	30.95	(1)	62.16	(7)	278.72	(1)	**2.65**	**(7)**	17.88	(6)
25	ghoulomb	o	10	0.00	(0)	0.00	(0)	246.67	(1)	**2512.92**	**(8)**	2545.02	(6)
26	gridColoring	o	5	0.48	(1)	152.71	(3)	0.51	(1)	0.10	(1)	**38.15**	**(3)**
27	rcpsp-max	o	10	42.11	(2)	0.00	(0)	30.00	(1)	596.41	(4)	**475.14**	**(4)**
28	solbat	s	15	746.31	(10)	1300.54	(14)	1540.71	(10)	311.92	(11)	**141.48**	**(15)**
29	sugiyama2	o	5	308.31	(5)	37.00	(5)	510.72	(5)	520.88	(5)	**8.64**	**(5)**
30	wwtp-random	s	5	0.03	(1)	322.21	(3)	2.79	(2)	0.00	(0)	**15.21**	**(5)**
31	wwtp-real	s	5	0.08	(3)	1239.45	(4)	0.31	(3)	71.83	(4)	**14.47**	**(5)**
32	bacp	o	15	847.78	(10)	1170.15	(5)	976.35	(10)	**28.54**	**(15)**	64.17	(15)
	Total		294	9881	(190)	15215	(124)	12866	(166)	7859	(185)	**12137**	**(212)**

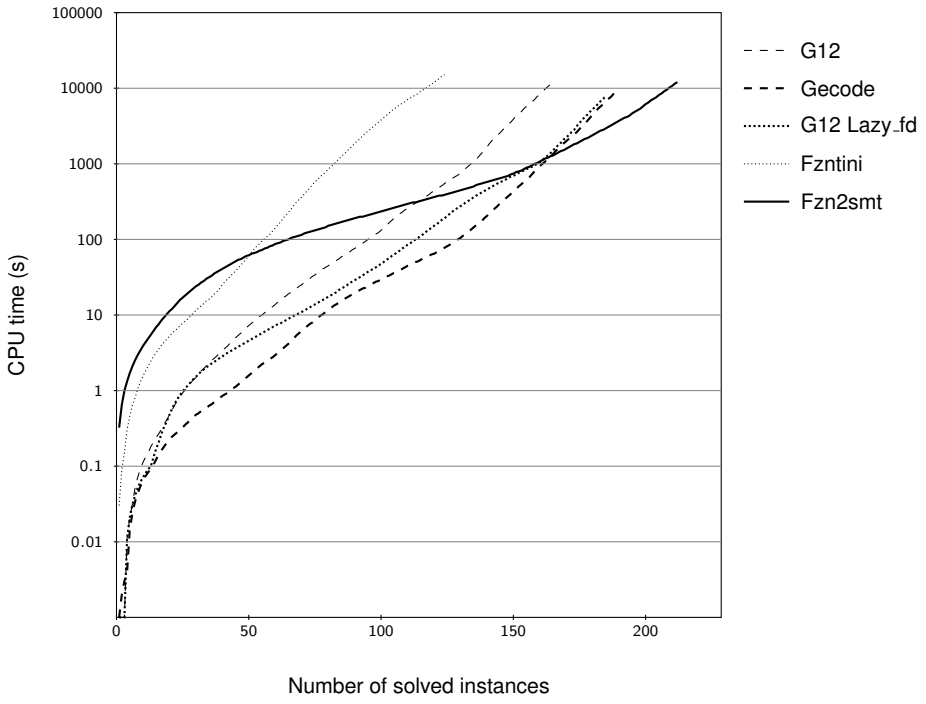


Figure 5.9: Number of solved instances and elapsed times (referred to Table 5.5).

from specific algorithms provided by the solvers.

Table 5.6 shows the performance of two FLATZINC solvers with support for global constraints compared to the performance of themselves, and that of `fzn2smt`, without using that support, on problems where global constraints do occur. Note that `fzn2smt` does not provide any specific support for global constraints. We have not included the results for G12 Lazy_fd with global constraints, since it exhibited very similar performance.

Again we can see that `fzn2smt` overall offers a bit better performance than Gecode and G12, even when they are using global constraints. This is even more significant if we take into account that most of these problems are optimization ones, and we have naively implemented a search procedure to supply the lack of support for optimization of SMT solvers (see Subsection 5.4.4). However, Gecode is best in 9 problems, whereas `fzn2smt` is best only in 8. We believe that unit propagation and conflict-driven lemma learning at Boolean level, partially compensate for the lack of specialized algorithms for global constraints in SMT solvers.

Table 5.6: Performance comparison of fzn2smt vs available FLATZINC solvers with global constraints.

			Gecode				G12				`fzn2smt`	
Problem	Type	#	+gc		−gc		+gc		−gc			
debruijn-binary	s	11	**4.14**	**(7)**	4.46	(6)	35.93	(7)	31.30	(6)	0.50	(1)
pentominoes	s	7	**65.82**	**(7)**	601.82	(7)	847.11	(4)	848.17	(4)	168.47	(2)
quasigroup7	s	10	**250.49**	**(6)**	278.73	(6)	1.55	(5)	1.72	(5)	20.03	(5)
rcpsp	o	10	10.56	(5)	12.07	(5)	0.51	(4)	97. 27	(5)	**993.74**	**(9)**
black-hole	s	10	20.88	(7)	69.68	(7)	2405.38	(6)	2423.48	(6)	**892.46**	**(8)**
nonogram	s	10	**493.61**	**(8)**	1353.63	(8)	351.35	(2)	336.93	(2)	1546.56	(7)
open-stacks	o	10	**168.56**	**(8)**	169.74	(8)	283.52	(8)	299.55	(8)	707.25	(7)
p1f	o	10	**730.60**	**(10)**	2.57	(8)	1.87	(8)	2.04	(8)	126.01	(9)
rect-packing	s	10	132.44	(6)	134.36	(6)	7.71	(5)	339.80	(6)	**106.66**	**(10)**
roster-model	o	10	**0.88**	**(10)**	1.04	(10)	4. 49	(10)	4.46	(10)	50.38	(10)
costasArray	s	5	**615.51**	**(4)**	943.75	(4)	411.82	(3)	423.09	(3)	664.13	(2)
depot-placement	o	15	1035.72	(12)	1205.75	(12)	519.64	(8)	522.81	(8)	**295.78**	**(15)**
filter	o	10	31.15	(1)	30.95	(1)	280.57	(1)	278. 72	(1)	**17.88**	**(6)**
ghoulomb	o	10	**1044.25**	**(10)**	0.00	(0)	598.54	(3)	246.67	(1)	2545.02	(6)
rcpsp-max	o	10	5.04	(2)	42.11	(2)	116.40	(2)	30. 00	(1)	**475.14**	**(4)**
sugiyama2	o	5	310.94	(5)	308.31	(5)	510.84	(5)	510. 72	(5)	**8.64**	**(5)**
bacp	o	15	848.88	(10)	847.78	(10)	979.85	(10)	976.35	(10)	**64.17**	**(15)**
Total		168	5769	(118)	6006	(105)	7357	(91)	7373	(89)	**8682**	**(121)**

+gc stands for using global constraints, and −gc stands for not using global constraints.

5.6 Impact of the Boolean Component

In this section we check the impact of the Boolean component of the instances in the performance of `fzn2smt`. We statistically compare the performance of `fzn2smt` with the best of the other available FLATZINC solvers, that is Gecode. We compare the number of solved instances by Gecode and `fzn2smt`, taking into account their Boolean component. In particular, we consider the number of Boolean variables and the number of non-unary clauses of the SMT instances resulting from the translation of each FLATZINC instance. We first look at this relation graphically (figure 5.10 and 5.11) and propose the following hypothesis: the more Boolean component the problem has, the better the performance of `fzn2smt` is with respect to that of Gecode. This hypothesis seems quite reasonable, because having a greater Boolean component, the SMT solver can better profit from built-in techniques such as unit propagation, learning and backjumping. We provide statistical tests to support this hypothesis.

First of all, we define the *normalized difference of solved instances* of each problem

$$dif = \frac{\#\texttt{fzn2smt}\ solved\ instances - \#Gecode\ solved\ instances}{\#instances}.$$

This difference ranges from -1 to 1, where -1 means that Gecode has solved all the instances and `fzn2smt` none, and 1 means the inverse.

We define the *Boolean variables ratio* r_v of each problem as the average of the number of Boolean variables divided by the number of variables of each SMT instance.

Similarly, we define the *disjunctions ratio* r_d of each problem as the average of the number of non-unary clauses divided by the number of constraints of each SMT instance.

In figure 5.10 we plot the differences with respect to the Boolean variables ratio, and in figure 5.11 with respect to the disjunctions ratio. These figures show that, the more Boolean variables and disjunctions the problem has, the better performance `fzn2smt` has, compared to Gecode. In particular, when the Boolean variables ratio r_v is above 0.2, `fzn2smt` is able to solve more instances than Gecode (i.e., the difference is positive). Only in two of those problems Gecode is able to solve more instances than `fzn2smt`, namely in **nmseq** (problem #2) and **open-stacks** (problem #14). In these problems, Boolean variables are mainly used in `bool2int()` constraints, hence these variables provide little Boolean structure and the SMT solver cannot profit from their contribution. When considering the disjunctions ratio, `fzn2smt` outperforms Gecode only when r_d is above 0.4. An exception to this fact is again on **nmseq**, where most disjunctions come from `bool2int()`.

Note that `fzn2smt` is able to solve more instances than Gecode in **propagation stress** (problem #16), which has neither Boolean variables nor disjunctions. This is probably due to the fact that the linear constraints of the problem can be solved by the *In-*

teger Difference Logic (IDL), a subset of LIA which is very efficiently implemented by Yices [DdM06b].

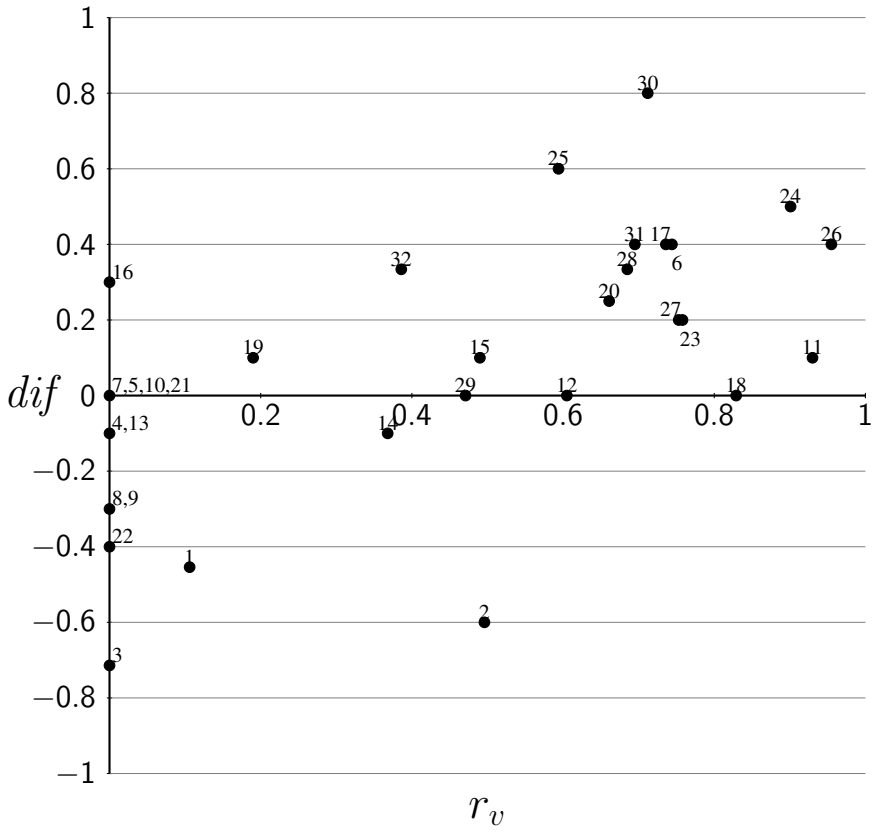


Figure 5.10: Normalized difference of solved instances between fzn2smt and Gecode with respect to the ratio of Boolean variables.

The numbers next to the points denote instance numbers (see Table 5.5).

We use a paired t-test in order to show that our method (`fzn2smt` with Yices) solves significantly more instances than Gecode. For each problem $i \in 1..n$, being n the number of considered problems, we take X_i as the normalized number of instances solved by `fzn2smt` and Y_i as the normalized number of instances solved by Gecode, and define $D_i = X_i - Y_i$ with null hypothesis

$$H_0 : \mu_D = \mu_x - \mu_y = 0$$

i.e., $H_0 : \mu_x = \mu_y$. Then we calculate the t-value as

$$t = \frac{\overline{D}_n}{S^*_D/\sqrt{n}}$$

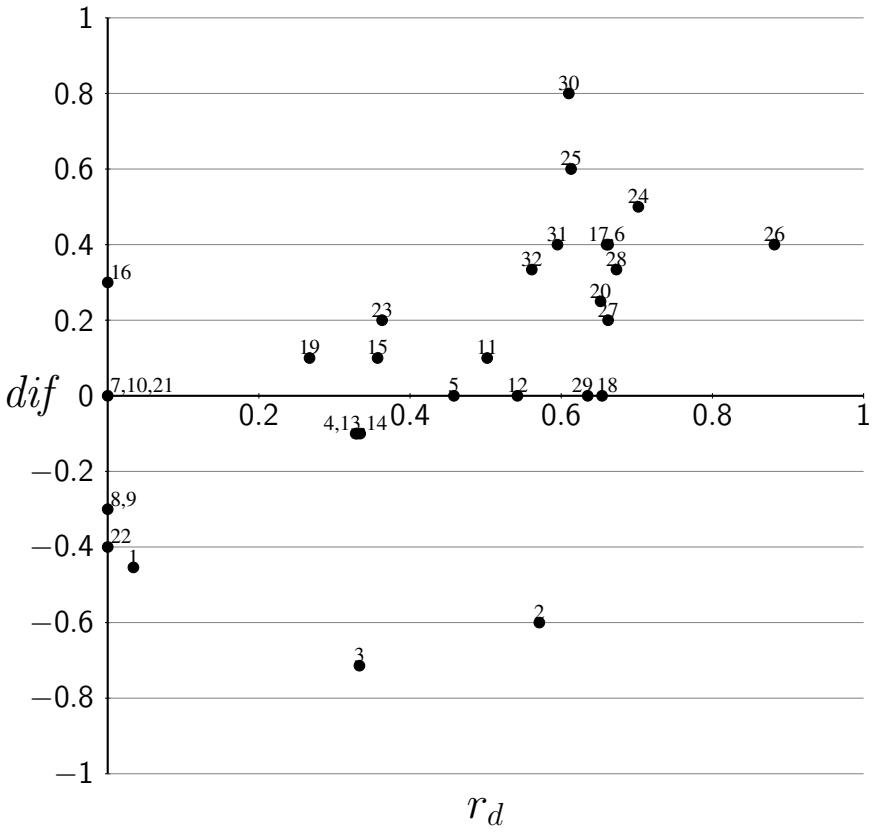


Figure 5.11: Normalized difference of solved instances between fzn2smt and Gecode with respect to the ratio of disjunctions.

The numbers next to the points denote instance numbers (see Table 5.5).

where $\overline{D}_n$ is the sample mean of the D_i and S_D^* is the sample standard deviation of the D_i. This statistic follows a Student's-t distribution with $n-1$ degrees of freedom.

Table 5.7 shows that in the general case with all 32 problems there is no significant difference between the means of the two solvers (the probability of the null hypothesis is $p = 0.2354$). Therefore we cannot say that `fzn2smt` is statistically better than Gecode. But, already for problems with Boolean variables ratio $r_v \geq 0.1$ we observe a significant difference (i.e., with $p < 0.05$ in all tests) in favor of `fzn2smt`. This confirms our hypothesis: the higher the Boolean variables ratio is, the better the performance of `fzn2smt` is with respect to that of Gecode. We also note that if we use the disjunctions ratio r_d for comparing the means, the results are similar: with $r_d >= 0.4$ the difference of means is significant in favor of `fzn2smt`.

Table 5.8 shows that the difference of means of `fzn2smt` and G12 Lazy_fd is less significant. We have not taken into account the problems not supported by G12 Lazy_fd

due to unbounded integer variables. These results suggest that the two approaches work similarly well for the same kind of problems.

Table 5.7: Paired t-test I.

r_v	#problems	p
≥ 0.0	32	**0.2354**
≥ 0.1	21	**0.0150**
≥ 0.2	19	**0.0043**
≥ 0.3	19	**0.0043**
≥ 0.4	17	**0.0057**
≥ 0.5	14	**0.0001**
≥ 0.6	13	**0.0003**

r_d	#problems	p
≥ 0.0	32	**0.2354**
≥ 0.1	24	**0.0472**
≥ 0.2	24	**0.0472**
≥ 0.3	23	**0.0540**
≥ 0.4	17	**0.0060**
≥ 0.5	16	**0.0056**
≥ 0.6	11	**0.0006**

Paired t-test, with probability p of the null hypothesis, for the difference in mean of the number of solved instances by `fzn2smt` and Gecode, for problems with different ratios r_v and r_d.

Table 5.8: Paired t-test II.

r_v	#problems	p
≥ 0.0	27	**0.8929**
≥ 0.1	16	**0.0157**
≥ 0.2	15	**0.0152**
≥ 0.3	15	**0.0152**
≥ 0.4	14	**0.0147**
≥ 0.5	13	**0.0140**
≥ 0.6	12	**0.0028**

r_d	#problems	p
≥ 0.0	27	**0.8929**
≥ 0.1	20	**0.1853**
≥ 0.2	20	**0.1853**
≥ 0.3	19	**0.1856**
≥ 0.4	15	**0.0279**
≥ 0.5	14	**0.0274**
≥ 0.6	10	**0.0633**

Paired t-test, with probability p of the null hypothesis, for the difference in mean of the number of solved instances by `fzn2smt` and G12 Lazy_fd, for problems with different ratios r_v and r_d.

5.7 Summary

This chapter provides one of the main contributions of this thesis. Thanks to the experiments that we have conducted we have been able to prove that SMT is a very good general approach to solve CSPs and COPs.

First we designed a new language, called `Simply`, to easily specify combinatorial problems. This system translates `Simply` instances into SMT and solves them using state-of-the-art SMT solvers, with very acceptable solving times.

Subsequently, to compare different solvers on the same models, we have developed `fzn2smt`. In this case we have coded all types of variables (integers, floats, Booleans, arrays and sets) and all FLATZINC (the low-level language of MINIZINC) constraints. The performance of this approach is very competitive, equivalent or superior to the other approaches used in the experiment. We have also provided some arguments on why we have chosen some theories instead of others and on the nature of the encodings used. We have statistically shown that, the greater is the Boolean component of the problem, the better is the performance of `fzn2smt` with respect to other solvers. Finally we have implemented several strategies for dealing with optimization (linear and hybrid) with similar results to those obtained using dichotomic search.

Chapter 6

Weighted CSP and Meta-Constraints

In this chapter we first introduce `WSimply`, an extension of `Simply`. `WSimply` is a new framework for modelling and solving Weighted Constraint Satisfaction Problems (WCSP) using SMT technology. In contrast to other well-known approaches designed for extensional representation of goods or no-goods, and with no many declarative facilities, our approach aims to follow an intensional and declarative syntax style. In addition, our language has built-in support for some meta-constraints such as *priority* and *homogeneity*, which allow the user to easily specify rich requirements on the desired solutions, such as preferences and fairness.

Next, we present the system with their phases of preprocessing, compilation and solving. There is a special emphasis on the solving process. We propose two alternative strategies for solving these WCSP instances using SMT: by reformulating them into plain SMT and solving them by means of search algorithms, or by reformulating them into WSMT and solving them using UNSAT based algorithms.

Then, we provide the results of some experimentation showing the good performance of solving WCSP with SMT, and finally we describe an encoding of extensional WCSPs into SMT.

The content of this chapter is part of our publications:

- *A Proposal for Solving Weighted CSPs with SMT* [ABP+11a],
- *W-MiniZinc: A Proposal for Modeling Weighted CSPs with MiniZinc* [ABP+11c], and
- *Solving Weighted CSPs with Meta-Constraints by Reformulation into Satisfiability Modulo Theories* accepted with minor changes.

The objectives of the thesis achieved in this chapter are the second one (to prove

that using an SMT solver in conjunction with appropriate algorithms can be a robust approach for optimization variants of CSP) and specially the third one (to develop a system supporting meta-constraints, allowing the user to model Weighted CSP intensionally, and to solve them using SMT). The contributions described in this chapter range from the fifth to the seventh.

6.1 State-of-the-Art

There exist several systems for specification and solving of extensional WCSPs (see [DGSA]). Nevertheless, as far as we know, there does not exist any system supporting the intensional modelling of WCSPs, so we believe this thesis is the first work in this direction. We present `WSimply`[1] an extension of `Simply` for filling this gap, using reformulation into SMT.

Our system supports meta-constraints [PRB00], i.e., constraints on (soft) constraints. Meta-constraints can be very helpful in the modelling process, since they allow us to abstract to a higher level, expressing, e.g., priorities between a set of soft constraints, different levels of preference (multi-objective optimization), etc. Hence, the inclusion of meta-constraints increases the capability to easily model several real-world problems.

6.2 WSimply

`Simply` [BPSV09], as explained in the Section 5.2, is a declarative programming system for easy modelling and solving of CSPs. `Simply` essentially translates CSP instances (written in its own language) into SMT instances, which are fed into an SMT solver. If the SMT solver is able to find a solution, then this solution is translated back to a solution of the original CSP instance. Currently, `Simply` is integrated with the Yices SMT solver [DdM06b], using its built-in API. However, it could be easily adapted to work with other SMT solvers, either using external files or any other API. The language has the useful declarative facility of list comprehension, which allows for concise and elegant modelings.

Figure 24 shows how to model an instance of the NRP with `Simply` and the SMT formula to which it is finally translated. As we can see, the SMT formula has no solution due to the preferences of the nurses that we have defined. The typical approach is to declare as soft constraints these preferences.

Example 24 *Consider a simple instance of the NRP (details on the NRP are given in*

[1] `Simply` and `WSimply` are available at `http://ima.udg.edu/recerca/lap/simply/`

Section 6.4) with two shifts per day and two available nurses. Each shift must be covered with exactly one nurse. We also want to satisfy the preferences of the nurses. Say nurse 1 wants to work both days on shift 1, while nurse 2 wants to work the first day on shift 2 and the second day on shift 1.

This particular instance can be modelled with `Simply` *as shown in Figure 6.1. We can identify four sections in the* `Simply` *model: data, domains, variables and constraints. The data section allows us to define a particular instance of the problem. This section can also defined in a separate file. The variables section declares a bidimensional array of integer variables* `nd`*, where the size of the first dimension corresponds to the number of nurses, and the size of the second dimension to the number of days. The domain of the integer variables of this array is restricted to the range specified by* `dshifts`*, i.e., the possible shifts. In the constraints section, we first introduce the cover constraints: for each day and shift, we have to meet the cover requirements (one nurse in this case). To post this constraint we generate the list of nurses working every day* `d` *and shift* `st` *with a list comprehension* `[nd[n, d] | n in [1..nurses]]`*, and we use the global constraint* `Count` *to restrict the number of nurses working that day and shift to be one. Finally, we add the nurse preference constraints for each day, specifying which shift they prefer.*

```
Problem:nrp
   Data
         int nurses = 2;
         int days = 2;
         int shifts = 2;
   Domains
         Dom dshifts = [1..shifts];
   Variables
         IntVar nd[nurses, days] :: dshifts;
   Constraints
         % Covers
         Forall(d in [1..days], st in [1..shifts]) {
            Count([nd[n, d] | n in [1..nurses]], st, 1);
         };
         % Preferences
         nd[1,1] = 1;
         nd[1,2] = 1;
         nd[2,1] = 2;
         nd[2,2] = 1;
```

Figure 6.1: A Simply instance.

```
; The logic to be used
(set-logic QF_LIA)

; Variable declarations
(declare-fun nd_1_1 () Int)
(declare-fun nd_1_2 () Int)
(declare-fun nd_2_1 () Int)
(declare-fun nd_2_2 () Int)

; Bounds on the domain of the variables
(assert (and (<= nd_1_1 2) (>= nd_1_1 1)))
(assert (and (<= nd_1_2 2) (>= nd_1_2 1)))
(assert (and (<= nd_2_1 2) (>= nd_2_1 1)))
(assert (and (<= nd_2_2 2) (>= nd_2_2 1)))

; Cover constraints
(assert
  (and
    (= (+ (ite (= nd_1_1 1) 1 0) (ite (= nd_2_1 1) 1 0)) 1)
    (= (+ (ite (= nd_1_1 2) 1 0) (ite (= nd_2_1 2) 1 0)) 1)
    (= (+ (ite (= nd_1_2 1) 1 0) (ite (= nd_2_2 1) 1 0)) 1)
    (= (+ (ite (= nd_1_2 2) 1 0) (ite (= nd_2_2 2) 1 0)) 1)
  )
)

; Preference constraints
(assert (= nd_1_1 1))
(assert (= nd_1_2 1))
(assert (= nd_2_1 2))
(assert (= nd_2_2 1))

(check-sat)
```

Figure 6.2: SMT instance obtained from the Simply instance in Figure 6.1.

`Simply` translates the previous instance into the standard SMT-LIB v2 language [BST10b] as shown in Figure 6.2. First of all we need to specify the background theory to be used. In this example we use Linear Integer Arithmetic (LIA). Then, we define the variables of the problem as integer. Next, we use the `assert` *operator to post the constraints, which are described in prefix notation. The first set of constraints bounds*

the domain of the variables. The second set corresponds to the translation of the cover constraints. Here, we use the operator `ite` *(if-then-else, or conditional expression). For example, the expression* `(ite (= nd_1_1 1) 1 0)` *evaluates to 1 when* `nd_1_1` *is equal to 1, and to 0 otherwise. Each of the four lines with the* `ite` *operator guarantee that there is exactly one nurse per shift. Finally, we add the translation of the preference constraints.*

Here we detail the extensions introduced to `Simply` in order to allow for soft constraints. The new language and system is called `WSimply`. The basic type of soft constraint in `WSimply` is of the form:

$$(constraint)\,@\,\{expression\};$$

where the value of $expression$ is the weight (cost) of falsifying the associated $constraint$. The parentheses around the constraint and the curly brackets around the weight expression are optional. The expression must be a linear integer arithmetic expression.[2] It can either be valuable at compile time, or contain decision variables. Weight expressions should always evaluate to a non-negative integer, as they amount to a cost. When negative, falsifying the constraint will be considered zero cost.

Example 25 *The instance in Example 24 has no solution. Hence, we could wish to model the preferences as soft constraints (and give, e.g., double importance to the preferences of nurse 1). This could be modelled in* `WSimply` *by replacing the preference constraints with the following:*

```
% Preferences
(nd[1,1] = 1) @ 2;
(nd[1,2] = 1) @ 2;
(nd[2,1] = 2) @ 1;
(nd[2,2] = 1) @ 1;
```

Most, if not all, existing WCSP solving systems consider an extensional approach, i.e., deal with instances consisting of an enumeration of good/no-good tuples for hard constraints, and no-good tuples with an associated cost for soft constraints.

Our proposal follows the other direction and aims to allow the user to model soft constraints in intension. Consider for instance two variables x and y, with domain $\{1, 2\}$, and the soft constraint $x < y$ with falsification cost 1. In an extensional approach, we would model this problem with the following soft no-good tuples: $(x = 1, y = 1, 1), (x =$

[2] We restrict to linear expressions, since we rely on SMT solvers and only a few of them incorporate (some limited) support for non-linear expressions.

$2, y = 1, 1)$ and $(x = 2, y = 2, 1)$. In `WSimply` we would express this with the soft constraint `(x < y) @ {1};`.

Degree of Violation

An interesting detail to remark is that by allowing the use of decision variables in the cost expression we can encode the degree of violation of a constraint. For instance, for a nurse working more than five turns in a week, the violation cost could be increased by one unit for each extra worked turn:

$$(\texttt{worked\_turns} < 6) \,@\, \{\texttt{base\_cost} + \texttt{worked\_turns} - 5\};$$

Labeled Constraints

Soft constraints can be labeled as follows (again, the parentheses around the constraint and the curly brackets around the weight expression are optional):

$$\#label : (constraint) \,@\, \{expression\};$$

The labels can be used to refer to the respective constraints in other (either soft or hard) constraints. For example:

```
#A: (a>b)@1;
#B: (a>c)@2;
#C: (a>d)@1;
(Not A And Not B) Implies C;
```

Labels are also used in meta-constraints, which we introduce in the next subsection. Moreover, indexed labels are supported, as we show in Section 6.4.1. This allows for convenient modellings in many cases.

6.3 Meta-Constraints

In order to provide a higher level of abstraction in the modelling of over-constrained problems, in [PRB00] several meta-constraints are proposed. By meta-constraint we refer to a constraint on constraints. Meta-constraints allow us to go one step further in the specification of the preferences on the soft constraint violations. `WSimply` covers all meta-constraints introduced in [PRB00], plus several variants and alternative meta-constraints, that we group in the following three families:

1. **Priority.** The user may have some preferences which soft constraints to violate. For instance, if there is an activity to perform and worker 1 *doesn't want* to perform it while worker 2 *should not* perform it, then it is better to violate the first constraint than the second. It would be useful to free the user of deciding the exact value of the weight of each constraint. To this end, we allow the use of undefined weights, denoted by "_":

$$\#label : (constraint) \,@\, \{\_\};$$

The value of this undefined weight is computed at compile time according to the priority meta-constraints that refer to the label. This simplifies the modelling of the problem, since the user does not need to compute any concrete weight. `WSimply` provides the following meta-constraints related to priority:

- `samePriority(`$List$`)`, where $List$ is a list of labels of soft constraints. This meta-constraint gives the same priority, i.e., the same weight, to the constraints denoted by the labels in $List$.
- `priority(`$List$`)`, where $List$ is a list of labels of soft constraints. This constraint orders the constraints denoted by the labels in $List$ by decreasing priority. In other words, it imposes decreasing weights.
- `priority(`$label_1$`,`$label_2$`,`n`)`, with $n > 1$, defines how many times it is worse to violate the constraint corresponding to $label_1$ than to violate the constraint corresponding to $label_2$. That is, if $weight_1$ and $weight_2$ denote the weights associated with $label_1$ and $label_2$, respectively, it states $weight_1 \geq weight_2 * n$.
- `multiLevel(`*ListOfLists*`)`, where *ListOfLists* is a list of lists of labels. This meta-constraint states that the weight of each one of the constraints (denoted by the labels) in each list is greater than the aggregated weight of the constraints in the following lists. For example,

```
multiLevel([[A,B,C],[D,E,F],[G,H,I]]);
```

states that the cost of falsifying each of the constraints (denoted by) `A`, `B` and `C` is greater than the cost of falsifying `D`, `E`, `F`, `G`, `H`, and `I` together and, at the same time, the cost of falsifying each of the constraints `D`, `E` and `F` is greater than the cost of falsifying `G`, `H`, and `I` together.

2. **Homogeneity.** The user may wish that there is some homogeneity in the amount of violation of disjoint groups of constraints. For instance, for the sake of fairness, the number of violated preferences of nurses should be as homogeneous as possible. `WSimply` provides the following meta-constraints related to homogeneity:

- `atLeast(`$List$`,`p`)`, where $List$ is a list of labels of soft constraints and p is a positive integer in $1..100$. This meta-constraint ensures that the percentage of constraints denoted by the labels in $List$ that are satisfied is at least p.
- `homogeneousAbsoluteWeight(`$ListOfLists$`,`v`)`, where $ListOfLists$ is a list of lists of labels of soft constraints and v is a positive integer. This meta-constraint ensures that, for each pair of lists in $ListOfLists$, the difference between the cost of the violated constraints in the two lists is at most v. For example, given

  ```
  homogeneousAbsoluteWeight([[A,B,C],[D,E,F,G]],10);
  ```

  if the weights of constraints `A`, `B` and `C` are 5, 10 and 15 respectively, and constraints `A` and `B` are violated and constraint `C` is satisfied, then the cost of the violated constraints in `[D,E,F,G]` must be between 5 and 25.
- `homogeneousAbsoluteNumber(`$ListOfLists$`,`v`)`. Same as above, but where the maximum difference v is between the number of violated constraints.
- `homogeneousPercentWeight(`$ListOfLists$`,`p`)`, where $ListOfLists$ is a list of lists of labels of soft constraints and p is a positive integer in $1..100$. This meta-constraint is analogous to `homogeneousAbsoluteWeight`, but where the maximum difference p is between the percentage in the cost of the violated constraints (with respect to the cost of all the constraints) in each list.
- `homogeneousPercentNumber(`$ListOfLists$`,`p`)`. Same as above, but where the maximum difference p is between the percentage in the number of violated constraints.

  We remark that the meta-constraints `homogeneousAbsoluteWeight` and `homogeneousPercentWeight`, are not allowed to refer to any constraint with undefined weight. This is because, as said, constraints with undefined weight are referenced by priority meta-constraints, and their weight is determined at compile time accordingly to those priority meta-constraints, independently from other constraints. Hence, since the `homogeneousAbsoluteWeight` and `homogeneousPercentWeight` meta-constraints also constrain the weight of the referenced constraints, if they were allowed to reference constraints with undefined weights, this could lead to incompleteness.

3. **Dependence.** Particular configurations of violations may entail the necessity to satisfy other constraints, that is, if a soft constraint is violated then another soft constraint must not be violated, or a new constraint must be satisfied. For instance, in the context of the NRP, we can imagine that working the first or the last turn of the day is penalized and, if somebody works in the last turn of one day, then he cannot work in the first turn the next day. This could be succinctly stated as follows:

```
#A: not_last_turn_day_1@w1;
#B: not_first_turn_day_2@w2;
(Not A) Implies B;
```

stating that, if constraint `A` is violated, then constraint `B` becomes mandatory.

Although the priority meta-constraints are discussed in [PRB00], they are not really developed into detail, and the multilevel meta-constraint is not considered. Our `atLeast` homogeneity meta-constraint subsumes the homogeneity meta-constraint defined in [PRB00], while the `homogeneousAbsoluteWeight`, `homogeneousAbsoluteNumber`, `homogeneousPercentWeight` and `homogeneousPercentNumber` meta-constraints are new. The dependence meta-constraints are the same as in [PRB00].

6.4 Modelling Example

In this section we illustrate the use of `WSimply` meta-constraints on a paradigmatic example of over-constrained CSP: the Nurse Rostering Problem (NRP). In a NRP we have to generate a roster assigning shifts to nurses over a period of time subject to a number of constraints. These constraints, that can be hard or soft, are usually defined by regulations, working practices and nurse preferences [MBL09].

We choose a simplified variant of the GPost NRP instance.[3] In this example we consider 4 weeks (28 days), 8 nurses (4 full-timers and 4 part-timers) and two shift types (day and night). We define an array variable `sh[8,28]` with domain `[0..2]`, where `sh[i,d] = 0` means that the `i`-th nurse does not work on day `d`, `sh[i,d] = 1` means that the nurse works on the day shift of day `d`, and `sh[i,d] = 2` means that the nurse works on the night shift of day `d`. Nurses numbered from 1 to 4 are full-timers, and from 5 to 8 are part-timers.

Full-timers work exactly 18 shifts in 4 weeks, while part-timers work only 10. We can encode this as follows:

```
Forall(i in [1..8]) {
 If (i<5) Then {Count([sh[i,d]>0|d in [1..28]],True,18);}
 Else {Count([sh[i,d]>0|d in [1..28]],True,10);};
};
```

Each nurse works at most 4 night shifts, of which at most 3 are consecutive. In order to refer to the number of night shifts per worker we have to introduce another array variable `tns[8]` with domain `[0..4]`:

[3] `http://www.cs.nott.ac.uk/~tec/NRP/`

```
Forall(i in [1..8]) {
 Count([sh[i,d]|d in [1..28]],2,tns[i] );
};
 % restrict the number of consecutive night shifts
Forall(i in [1..8], d in [1..25]) {
 ((sh[i,d]>1) And (sh[i,d+1]>1) And (sh[i,d+2]>1))
 Implies Not(sh[i,d+3]>1);
};
```

Note that the maximum number of night shifts is bounded by the domain of the array `tns`. Moreover, since the domain of the integers in array `sh` is `[0..2]`, we could alternatively write, e.g., `sh[i,d] = 2` instead of `sh[i,d] > 1`. However, we have observed that using strict inequalities often results in better performance, possibly due to the special treatment given to them by the linear integer arithmetic solver integrated with Yices.

6.4.1 Soft Constraints

There is a penalty for a single night shift (for the sake of simplicity, we ignore the first and last days of the roster):

```
Forall(i in [1..8], d in [1..26]) {
 #NSP[i,d]: Not((sh[i,d]<2) And (sh[i,d+1]>1) And
                (sh[i,d+2]<2))@{_};
};
```

Note that we can introduce arrays of labels in the `Forall` statement, which are indexed according to the `Forall` variables. We leave the weights undefined, since we just want all of them be the same. To this end, we only need to post the following meta-constraint (which uses the indexed labels introduced in the `Forall` statement):

```
samePriority([NSP[i,d]|i in [1..8],d in [1..26]]);
```

Similarly, we want to penalize isolated free days (again, the first and last days of the roster are ignored for the sake of simplicity):

```
Forall(i in [1..8], d in [1..26] {
 #AFD[i,d]: Not((sh[i,d]>0) And (sh[i,d+1]<1) And
                (sh[i,d+2]>0))@{_};
};
samePriority([AFD[i,d]|i in [1..8],d in [1..26]]);
```

Since we consider that violating the AFD constraints is 10 times preferable than violating the NSP constraints, we use the following meta-constraint:

```
priority(NSP[1,1],AFD[1,1],10);
```

Note that it is enough to state this priority between the first constraints of each group, since all constraints of each group have the same priority.

A full-timer has to work 4 or 5 days per week. We want to consider the deviation from this number as the violation degree of the constraint. This can be expressed as follows, by introducing an array variable tw[4,4], where tw[i,j] denotes the number of working days for nurse i on week j:

```
Forall(i in [1..4]) {
 Count([sh[i,d]> 0|d in [1..7]],True,tw[i,1]);
 Count([sh[i,d]>0|d in [8..14]],True,tw[i,2]);
 ...
};
Forall(i in [1..4], w in [1..4]) {
 Not(tw[i,w]>5) @ {tw[i,w]-5};
 Not(tw[i,w]<4) @ {4-tw[i,w]};
};
```

Finally, we can use homogeneity meta-constraints in order to guarantee a minimum satisfaction on the free days assigned to each nurse. This can be achieved as follows. We first state, as a soft constraint of weight 1, each free day requested by each nurse being free. We assume that each nurse has asked for five preferred free days, which are stored in an input data array free[8,5].

```
Forall(i in [1..8], f in [1..5]) {
 #PFD[i,f]:(sh[i,free[i,f]]<1)@1;
};
```

Then, the following meta-constraints can be used in order to guarantee that, globally, 40% of the preferences of the nurses are satisfied and, at the same time, to homogeneously satisfy the preferences among nurses (the difference in the percentage of violated preferences for the different nurses is no more than 50):

```
atLeast([PFD[i,f]|i in [1..8],f in [1..5]],40);
homogeneousPercentNumber([[PFD[1,f]|f in [1..5]],
                [PFD[2,f]|f in [1..5]],...],50);
```

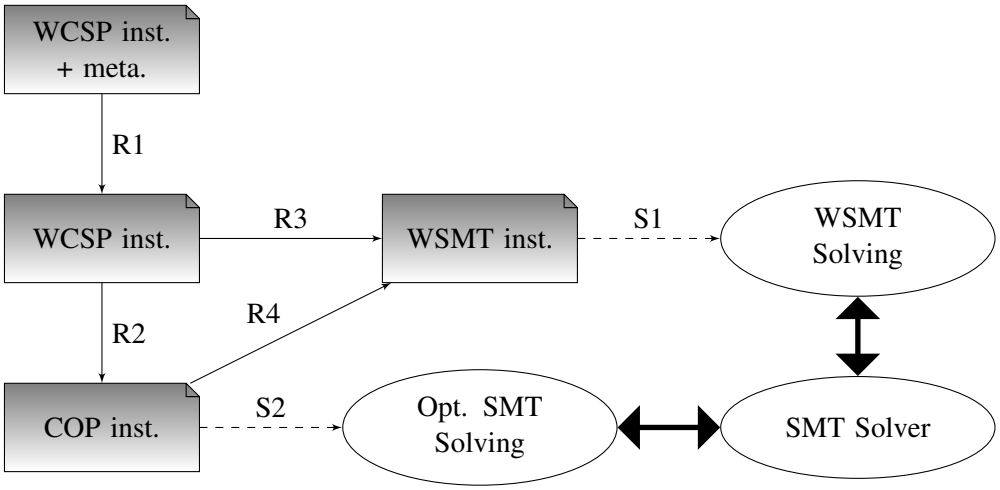


Figure 6.3: Basic architecture and solving process of WSimply.

Finally, we impose that it is ten times better not to have an isolated free day than resting one of the preferred days. This is also a requirement of the GPost instance:

```
priority(AFD[1,1],PFD[1,1],10);
```

It is worth noting that, at this point, the system is able to determine a concrete value for the undefined weights of the `AFD` and `NSP` constraints.

6.5 Solving Process

Figure 6.3 shows the basic architecture and solving process of `WSimply`. `WSimply` reformulates the input instance into the suitable format for the solving procedures. We have four reformulations: (R1) from a WCSP instance with meta-constraints into a WCSP instance (without meta-constraints), (R2) from a WCSP instance into a COP instance, (R3) from a WCSP instance into a WSMT instance and (R4) from a COP instance into a WSMT instance. We recall that the constraints in the previous instances (WCSP and COP) are expressed into the `WSimply`.

Once the problem has been properly reformulated, we can apply two different solving approaches: WSMT Solving (S1) or Optimization SMT Solving (S2).

In the following subsections we describe the different reformulations and solving procedures.

6.5.1 Reformulating WCSP with Meta-Constraints into WCSP (R1)

We remove all the meta-constraints by reformulating them into hard and soft `WSimply` constraints.

As we have showed in Section 6.2, meta-constraints use labels. Then, first of all, we introduce a new reification variable for each labelled soft constraint, of the form:

$$\#label : (constraint) \,@\, \{expression\};$$

and we replace the constraint by:

$$b_{label} \Leftrightarrow constraint;$$
$$b_{label} \,@\, \{expression\};$$

where b_{label} is a fresh (Boolean) reification variable.

In the following we show how `WSimply` reformulates the priority, homogeneity and dependence meta-constraints.

Reformulation of Priority Meta-Constraints

To deal with the *priority* meta-constraints, we create a system of linear inequations on the (probably undefined) weights of the referenced soft constraints. The inequations are of the form $w = w'$, $w > w'$ or $w \geq n \cdot w'$, where w is a variable, w' is either a variable or a non-negative integer constant, and n is a positive integer constant. For example, given

```
#A:(a>b)@{3};
#B:(a>c)@{_};
#C:(a>d)@{_};
#D:(c=2-x)@{_};
priority([A,B,C]);
priority(D,B,2);
```

the following set of inequations is generated:

$$w_A = 3, w_B > 0, w_C > 0, w_D > 0,$$
$$w_A > w_B, w_B > w_C,$$
$$w_D \geq 2 \cdot w_B$$

This set of inequations is fed into an SMT solver[4] which acts as an oracle at compile time, so that a model, i.e., a minimal value for the undefined weights satisfying the

[4]In fact, the set of inequations could be fed into any linear integer arithmetic solver.

inequations, can be found. Following the previous example, the SMT solver would return us a model such as, e.g.:

$$w_A = 3, w_B = 2, w_C = 1, w_D = 4$$

This allows the reformulation of the original problem into an equivalent WCSP without undefined weights:

```
#A:(a>b)@{3};
#B:(a>c)@{2};
#C:(a>d)@{1};
#D:(c=2-x)@{4};
```

Hence, with the meta-language, and thanks to this simple use of a solver as an oracle at compile time, we free the user of the tedious task of thinking about concrete weights for encoding priorities.

In the case of the `multiLevel` meta-constraint, given for example

```
#A:(a>b)@{_};
#B:(a>c)@{_};
#C:(a>d)@{_};
#D:(c=2-x)@{_};
multiLevel([[A,B][C,D]]);
```

the following set of inequations would be generated:

$$\begin{aligned} &w_A > 0, w_B > 0, w_C > 0, w_D > 0, \\ &w_A > (w_C + w_D), \\ &w_B > (w_C + w_D), \end{aligned}$$

and the SMT solver would return us a model such as, e.g.:

$$w_A = 3, w_B = 3, w_C = 1, w_D = 1$$

We remark that the weight expressions of the constraints referenced by a priority meta-constraint must either be undefined or evaluable at compile time, i.e., they cannot use any decision variable, since our aim is to compute all undefined weights at compile time. Moreover, if the set of inequations turns out to be unsatisfiable, the user will be warned about this fact during compilation.

Reformulation of Homogeneity Meta-Constraints

We reformulate the *homogeneity* meta-constraints by reifying the referenced constraints and constraining the number of satisfied constraints. For example, the meta-constraint `atLeast(`$List$`,`p`)` is reformulated into:

```
Count(ListReif, True, n);
n >= val;
```

where `ListReif` is the list of Boolean variables resulting from reifying the constraints referenced in $List$, and `val` is computed in compile time and is equal to $\lceil length(List) * p/100 \rceil$. `Count(`$l$`,`$e$`,`$n$`)` is a `Simply` global constraint that is satisfied if and only if there are exactly n occurrences of the element e in the list l.

The meta-constraint `homogeneousPercentWeight(`$ListOfLists$`,`p`)`, is reformulated into:

```
Sum([weight_label[1][j]|j in [1..len[1]],total_wei[1]);
Sum([If_Then_Else(ListOfLists[1][j])(0)(weight_label[1][j])
        | j in [1..len[1]]], vio_wei[1]);
(vio_wei[1]*100 Div total_wei[1])>=min_homogen;
(vio_wei[1]*100 Div total_wei[1])=<max_homogen;

...

Sum([weight_label[n][j]|j in [1..len[n]],total_wei[n]);
Sum([If_Then_Else(ListOfLists[n][j])(0)(weight_label[n][j])
        | j in [1..len[n]]], vio_wei[n]);
(vio_wei[n]*100) Div total_wei[n])>=min_homogen;
(vio_wei[n]*100) Div total_wei[n])=<max_homogen;
(max_homogen-min_homogen)<p;
```

where `len[i]` is the length of the `i`-th list in $ListOfLists$, `weight_label[i][j]` is the weight associated with the `j`-th label of the `i`-th list, `total_wei[i]` is the aggregated weight of the labels in the `i`-th list and `n` is the length of $ListOfLists$. Notice that according to the `Sum` constraints, `vio_wei[i]` denotes the aggregated weight of the violated constraints in the `i`-th list. Finally, `min_homogen` and `max_homogen` are fresh new variables.

Since we restrict to linear integer arithmetic, the `total_wei[i]` expressions must be evaluable at compile time. This requires the weights of the constraints referenced by this meta-constraint to be evaluable at compile time.

The reformulation of `homogeneousAbsoluteWeight(`*ListOfLists*`,`v`)` is analogous to the previous one, but where instead of computing the percentage on `vio_wei[i]`, we can directly state:

```
...
vio_wei[1]>=min_homogen;
vio_wei[1]=<max_homogen;
...
vio_wei[n]>=min_homogen;
vio_wei[n]=<max_homogen;
(max_homogen-min_homogen)=<v;
```

Our reformulation allows the meta-constraint `homogeneousAbsoluteWeight` to reference constraints whose weight expression uses decision variables and is not evaluable at compile time. However, as pointed out in Section 6.3, it cannot reference constraints with undefined ("_") weight.

The reformulations of the meta-constraints `homogeneousAbsoluteNumber` and `homogeneousPercentNumber` are similar to the previous ones, but where we count the number of violated constraints instead of summing their weights.

Reformulation of Dependence Meta-Constraints

The *dependence* meta-constraints are straightforwardly reformulated by applying the logical operators between constraints directly supported by `Simply` on the corresponding reification variables.

6.5.2 Reformulating WCSP into COP (R2)

In order to convert our WCSP instance into a COP instance, we firstly replace each soft constraint $C_i \,@\, w_i$ by the following constraints where we introduce a fresh integer variable o_i:

$$\neg(w_i > 0 \ \wedge \ \neg C_i) \ \rightarrow \ o_i = 0 \tag{6.1}$$

$$(w_i > 0 \ \wedge \ \neg C_i) \ \rightarrow \ o_i = w_i \tag{6.2}$$

If the weight expression, w_i, evaluates to a value less or equal than 0, then the cost of falsifying C_i is 0, otherwise it is w_i. Since we are defining a minimization problem we could actually replace Equation (6.1) by $o_i \geq 0$.

Secondly, we introduce another fresh integer variable O, which represents the sum of the o_i variables, i.e., the optimization variable of the COP to be minimized, and the following constraint:

$$O = \sum_{i=1}^{m} o_i \tag{6.3}$$

Finally, we keep the original hard constraints with no modification.

6.5.3 Reformulating WCSP into WSMT (R3)

To the best of our knowledge, existing WSMT solvers only accept WSMT clauses whose weights are constants. Therefore, we need to convert the WCSP instance into a WSMT instance where all the WSMT clauses have a constant weight.

We apply the same strategy as in R2, i.e., we firstly replace each soft constraint $C_i \,@\, w_i$, where w_i is not a constant (involves variables), by the constraints 6.1 and 6.2 introducing a fresh integer variable o_i. Secondly, we add the following set of soft constraints over each possible value of each o_i variable:

$$\bigcup_{v_j \in V(o_i)} o_i \neq v_j \,@\, v_j \tag{6.4}$$

where $V(o_i)$ is the set of all possible positive values of o_i. These values are determined by evaluating the expression for all the possible values of the variables, and keeping only the positive results. Notice that at this point we do have a WCSP instance where all the soft constraints have a constant weight.

Finally, we replace each soft constraint $C_i \,@\, w_i$, by the WSMT clause (C'_i, w_i) where C'_i is the translation of C_i into SMT as described in [BPSV09]. We also replace each hard constraint by its equivalent hard SMT clause.

6.5.4 Reformulating COP into WSMT (R4)

Taking into account that the optimization variable O of the COP instance is the integer variable that represents the objective function we only need to add the following set of WSMT clauses: $\bigcup_{i=1}^{i=W}(O < i, 1)$, where W is the greatest value the objective variable can be evaluated to. A more concise alternative could result from using the binary representation of W, i.e., adding the set of WSMT clauses $\bigcup_{i=0}^{i<\lceil log_2(W+1)\rceil}(\neg b_i, 2^i)$, and the hard clause $(\sum_{i=0}^{i<\lceil log_2(W+1)\rceil} 2^i \cdot b_i = O, \infty)$.

We finally replace all the constraints of the COP instance with the equivalent hard SMT clauses as described in [BPSV09].

6.5.5 Solving with SMT

From Figure 6.3 we see that currently we can apply two solving methods in `WSimply`: WSMT solving which receives as input a WSMT instance and Optimization SMT solving which receives as input a COP instance.

WSMT Solving (S1)

The SMT solver Yices [DdM06b] offers a non-exact algorithm[5] to solve WSMT instances. We refer to this solving method as yices. Since this is yet an immature research topic in SMT, we have extended the Yices framework by incorporating other exact algorithms from the MaxSAT field. There, we can find two main classes of algorithms: branch and bound based and UNSAT core based algorithms. The solvers that implement the latter clearly outperform branch and bound based solvers on industrial and some crafted instances, and constitute an emerging technology.

In the following we describe the basic scheme of UNSAT core based algorithms. A WSMT problem φ can be solved through the resolution of a sequence of SMT instances as follows. Let φ_k be an SMT formula that is satisfiable if, and only if, φ has an assignment with cost smaller than or equal to k (k plays the role of the bound that we impose on the objective function). If the cost of the optimal assignment to φ is k_{opt}, then the SMT problems φ_k, for $k \geq k_{opt}$, are satisfiable, while for $k < k_{opt}$ are unsatisfiable. Note that k may range from 0 to $\sum_{i=1}^{m} w_i$ (the sum of the weights of the soft clauses). When the weights are expressions rather than constants, we estimate an upper-bound. This is done by evaluating the expression for all the possible values of the variables. The search for the value k_{opt} can be done following different strategies; searching from $k = 0$ to k_{opt} (increasing k while φ_k is unsatisfiable); from $k = \sum_{i=1}^{m} w_i$ to some value smaller than k_{opt} (decreasing k while φ_k is satisfiable); or alternating unsatisfiable and satisfiable φ_k until the algorithm converges to k_{opt} (for instance, using a binary search scheme). The key point to boost the efficiency of these approaches is to know whether we can exploit any additional information from the execution of the SMT solver for the next runs.

Since `WSimply` is designed to use SMT solvers as a black box, UNSAT core based algorithms can be easily integrated into `WSimply`.

In particular, we have implemented the WPM1 algorithm from [ABL09, MSP09], which is based on the detection of unsatisfiable cores. These are UNSAT core based

[5] Non-exact algorithms do not guarantee optimality.

algorithms, where the parameter k ranges from 0 to k_{opt}. Then, for every UNSAT answer, they analyze the core of unsatisfiability of the formula returned by the SMT solver. This information is incorporated in the form of redundant clauses into the next call to the SMT solver which help to boost the propagation. In our experiments, we refer to the method which uses the WPM1 algorithm as core.

The pseudo-code of the WPM1 algorithm based on calls to an SMT solver is described in Algorithm 15. This algorithm is the weighted version of the FuMalik algorithm [ABL09, MSP09] for partial MaxSAT, see Algorithm 12. In those works, the underlying solver was a SAT solver. This is the first time SMT technology is incorporated in the implementation of Algorithm 15.

In Algorithm 15, we iteratively call an SMT solver with a weighted working formula φ, but excluding the weights. The SMT solver will say whether the formula is satisfiable or not (variable st) and in case the formula is unsatisfiable, it will give an unsatisfiable core (φ_c). When the SMT solver returns an unsatisfiable core, we compute the minimum weight of the clauses of the core (w_{min} in the algorithm). Then, we transform the working formula by duplicating the clauses in the core. Then, in one of the copies we give to the clauses their original weight minus the minimum weight. On the other copy, we extend the clauses with the blocking variables (BV in the code) and we give them the minimum weight. Finally, we add the cardinality constraint on the blocking variables using the standard encoding of the exactly_one Boolean constraint. Note that we could assert this cardinality constraint using the Linear Integer Arithmetic theory, however, the Boolean encoding has shown a better performance in our experiments. We finally add w_{min} to the $cost$.

Optimization SMT Solving (S2)

The optimization solving approach is the one described in Subsection 5.4.4. Nevertheless, currently `WSimply` only supports the binary bounding strategy of Algorithm 14. In fact, this has been the approach that has exhibited the best performance (see Table 5.4). In the following, we will refer to this solving method as dico.

6.6 Benchmarking

In order to show the usefulness of meta-constraints we have conducted several experiments on a set of instances of the *Nurse Rostering Problem* (NRP) and on a variant of the *Balanced Academic Curriculum Problem* (BACP).

Algorithm 15 WPM1 Algorithm for SMT

Input: $\varphi = \{(C_1, w_1), \ldots, (C_n, w_n)\}$: CNF formula
Output: Cost of φ

```
cost ← 0
while true do
  (st, φ_c) ← SMT_ALGORITHM({C_i | (C_i, w_i) ∈ φ})
  if st = SAT then
    return cost
  else
    BV ← ∅
    w_min ← min{w_i | C_i ∈ φ_c ∧ isSoft(C_i)}
    for all C_i ∈ φ_c do
      if isSoft(C_i) then
        b ← New_variable()
        φ ← φ \ {(C_i, w_i)} ∪ {(C_i, w_i − w_min)} ∪ {(C_i ∨ b, w_min)}
        {when w_i − w_min = 0 the clause (C_i, w_i − w_min) is not added}
        BV ← BV ∪ {b}
      end if
    end for
    if BV = ∅ then
      return UNSAT
    end if
    φ ← φ ∪ {(exactly_one({b | b ∈ BV}), ∞)}
    cost ← cost + w_min
  end if
end while
```

6.6.1 Nurse Rostering Problem

There exist many formalizations of the NRP [BCBL04]. In our experiments we have considered the GPost instance (of which we have modelled a variant in Section 6.4) as well as many instances from the Nurse Scheduling Problem Library (NSPLib) [VM07], by conducting a precise study on the effects of the homogeneity meta-constraints on them.

GPost NRP

In Section 6.4 we have already presented how to model a variant of the GPost NRP with `WSimply` including several meta-constraints. `WSimply` shows a reasonably good performance solving the original GPost instance, compared to the results on the same problem reported in [MBL09]. The authors report 8 seconds (2.83GHz Intel® Core™ 2 Duo)

for finding the optimal solution (cost 3) with an ad hoc search with CPLEX over a previously computed enumeration of all possible schedules for each nurse. They also report 234 seconds (2.8GHz Pentium IV) for finding a non optimal solution (cost 8) with their generic local search method (VNS/LDS+CP) based on neighborhoods plus an exploration of the search space with CP and soft global constraints.

With `WSimply`, we have been able to find the optimal solution with the three solving approaches[6] (using a 2.6GHz Intel® Core™ i5) taking 12.27 seconds with the yices solving approach, 31.21 seconds with the core solving approach, and 126.00 seconds with the dico solving approach.

NSPLib Instances

In order to evaluate the effects of the homogeneity meta-constraints on the quality of the solutions and the solving times, we have conducted an empirical study over some instances from the NSPLib. The NSPLib is a repository of thousands of NRP instances, grouped in different sets and generated using different complexity indicators: size of the problem (number of nurses, days or shift types), shifts coverage (distributions over the number of nurses needed) and nurse preferences (distributions of the preferences over the shifts and days). Details can be found in [VM07].

In order to reduce the number of instances to work with, we have focused on the N25 set, which contains 7920 instances. Since the addition of homogeneity meta-constraints in these particular instances significantly increases their solving time, we have ruled out the instances taking more than 60 seconds to be solved with `WSimply` without the meta-constraints. The final chosen set consists of 5113 instances. The N25 set has the following settings:

- Number of nurses: 25
- Number of days: 7
- Number of shift types: 4 (including the free shift)
- Shift covers: minimum number of nurses required for each shift and day.
- Nurse preferences: a value between 1 and 4 (from most desirable to less desirable) for each shift and day, for each nurse.

The NSPLib also has several complementary files with more precise information like minimum and maximum number of days that a nurse should work, minimum and maximum

[6] See Subsection 6.5.5.

number of consecutive days, etc. We have considered the most basic case (case 1) which only constrains that

- the number of working days of each nurse must be exactly 5.

With the previous information we propose the NRP modelling of Figure 6.4, where we have as hard constraints the shift covers and the number of nurse working days, and as soft constraints the nurse preferences.

In the following we report the results of several experiments performed with `WSimply` over the set of 5113 chosen instances of the NRP, using a cluster with nodes with CPU speed 1GHz and 500 MB of RAM, and with a timeout of 600 seconds. We tested the three solving approaches (dico, yices and core). The times appearing in the tables are for the core approach, which was the one giving best results on this problem.

Table 6.1 shows the results for the chosen 5113 instances from the N25 set without homogeneity meta-constraints.

	μ	σ	Time	Cost	Abs. diff.	Rel. diff.
Normal	9.67	1.83	6.46	241.63	7.71	9.42

Table 6.1: Results on 5113 instances from the N25 set, with soft constraints on nurse preferences (without meta-constaints). μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; Time: mean solving time (in seconds); Cost: mean optimal cost; Abs. diff.: mean of differences between maximal and minimal costs; Rel. diff.: mean of differences between relative percentual maximal and minimal costs.

We can observe that if we only focus on minimizing the cost of the violated constraints, we can penalize some nurses much more than others. For instance, we could assign the least preferred shifts to one nurse while assigning the most preferred shifts to others. From the results in Table 6.1, we observe that the mean of absolute differences is 7.71 while the mean cost per nurse is around 9.67, which shows that the assignments are not really fair. In order to enforce fairness, we can extend the model of Figure 6.4 by adding homogeneity meta-constraints over the soft constraints on nurse preferences, as shown in Figure 6.5.

Table 6.2 shows the results after adding to the model of Figure 6.4 the meta-constraint of Figure 6.5, with factor $\mathtt{F} = 5$, while Table 6.3 shows the results for the same meta-constraint with factor $\mathtt{F} = 10$. Notice that this factor represents the maximal allowed difference between the penalization of the most penalized nurse and the less penalized nurse. From Table 6.1 we know that the mean of these differences among the chosen NRP instances is 7.71. The first row (Absolute 5) shows the results for the solved instances (2478 out of 5113) within the timeout. The second row shows the results without

```
Problem:nrp
 Data
   int n_nurses;
   int n_days;
   int n_shift_types;
   int covers[n_days, n_shift_types];
   int prefs[n_nurses, n_days, n_shift_types];
   int min_turns;
   int max_turns;
 Domains
   Dom dshifts = [1..n_shift_types];
   Dom dturns  = [min_turns..max_turns];
   Dom dnurses = [0..n_nurses];
 Variables
   IntVar nurse_day_shift[n_nurses, n_days]::dshifts;
   IntVar nurse_working_turns[n_nurses]::dturns;
   IntVar day_shift_nurses[n_days, n_shift_types]::dnurses;
 Constraints
   %%% Every nurse must work only one shift per day.
   %%% This constraint is implicit in this modelling.

   %%% The minimum number of nurses per shift and day
   %%% must be covered. Variables day_shift_nurses[d,st]
   %%% will contain the number of nurses working for
   %%% every shift and day.
   Forall(d in [1..n_days], st in [1..n_shift_types]) {
     Count([nurse_day_shift[n,d]|n in [1..n_nurses]],st,
         day_shift_nurses[d,st]);
   };
   [day_shift_nurses[d,st]>=covers[d,st]|d in [1..n_days],
         st in [1..n_shift_types]];
   %%% Nurse preferences are desirable but non-mandatory.
   %%% Each preference is posted as a soft constraint with
   %%% its label (#prefs[n,d,st]) and a violation cost
   %%% according to prefs[n,d,st].
   Forall(n in [1..n_nurses],d in [1..n_days],st in
     [1..n_shift_types]) {
     #prefs[n,d,st]:(Not(nurse_day_shift[n,d]=st))
       @{prefs[n,d,st]};
   };
   %%% The minimum and maximum number of working days of
   %%% each nurse must be between bounds (i.e. the domain
   %%% of nurse_working_turns[n]).
   Forall(n in [1..n_nurses]) {
     Count([nurse_day_shift[n,d]<>n_shift_types
         |d in [1..n_days]],
           True, nurse_working_turns[n]);
   };
```

Figure 6.4: WSimply model for the NRP.

the meta-constraint, for the solved instances (i.e., it is like Table 6.1 but restricted to these 2478 instances).

As we can observe, we reduce the absolute difference average from 4.81 to 4.32, which is a bit more than 10%. In particular, we reduce the absolute difference between the most penalized nurse and the less penalized nurse in 892 instances out of 2478. In

```
%%% Ask for homogeneity with factor F, over the lists of
%%%  soft constraints on nurse preferences.
homogeneousAbsoluteWeight([[prefs[n,d,st]|d in [1..n_days],
   st in [1..n_shift_types]]|n in [1..n_nurses]],F);
```

Figure 6.5: WSimply constraints to add to the NRP model in order to ask for homogeneity with factor `F` in the solutions.

	μ	σ	Time	Cost	Cost (TO)	Abs. diff.	#improved
Absolute 5	8.92	0.96	43.28	222.89	272.93	4.32	892
Normal	8.80	1.07	5.98	220.03	261.94	4.81	-

Table 6.2: Results when adding the homogeneousAbsoluteWeight meta-constraint with factor 5.Statistics for the 2478 solved instances with a timeout of 600 seconds. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; Time: mean solving time (in seconds); Cost: mean optimal cost; Cost (TO): mean of best lower bounds for those instances that exceeded the timeout; Abs. diff.: mean of differences between maximal and minimal costs; #improved: number of instances with improved absolute difference.

contrast, the average penalization per nurse increases from 8.80 to 8.92, but this is just 1.36%. The average global cost also increases, but only from 220.03 to 222.89. Hence, it seems reasonable to argue that it pays off to enforce homogeneity in this setting, at least for some instances. However, when homogeneity is enforced the solving time increases, since the instances become harder (there are 2635 instances which could not be solved within the timeout).

The conclusion is that an homogeneity factor $\texttt{F} = 5$ may be too restrictive. Therefore, we repeated the experiment but with a factor $\texttt{F} = 10$. The results are shown in Table 6.3.

	μ	σ	Time	Cost	Cost (TO)	Abs. diff.	#improved
Absolute 10	9.32	1.46	13.86	233.01	285.83	6.29	377
Normal	9.31	1.47	6.16	232.83	280.40	6.35	-

Table 6.3: Results when adding the homogeneousAbsoluteWeight meta-constraint with factor 10. Statistics for the 4167 solved instances with a timeout of 600 seconds. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; Time: mean solving time (in seconds); Cost: mean optimal cost; Cost (TO): mean of best lower bounds for those instances that exceeded the timeout; Abs. diff.: mean of differences between maximal and minimal costs; #improved: number of instances with improved absolute difference.

In this case only 946 out of 5113 could not be solved within the timeout. Although fewer instances are improved (377) the difference in the solving time really decreases and the mean of the best lower bounds for the unsolved instances is closer to the optimal value of the original instances. This suggests that it is possible to find a reasonable balance between the quality of the solutions and the required solving time with respect to the original problem.

Depending on the preferences of the nurses, the absolute difference may not be a good measure to enforce homogeneity. Nurse preferences are weighted with a value between 1 and 4 (from most desirable to less desirable shifts). Imagine a nurse who tends to weight with lower values than another. Then, even if this nurse has many preferences unsatisfied, her total penalization could be lower than the one of other nurses with less unsatisfied preferences. Therefore, it seems more reasonable to compute the relative difference, as it allows to compare the relative degree of unsatisfied preferences.

Table 6.4 shows the results for the meta-constraint `homogeneousPercentWeight` with factor 6, which means that the relative percentual difference between the most penalized nurse and the less penalized nurse must be less than or equal to 6. The first row (Percent 6) shows the results for the solved instances (2109 out of 5113) within the timeout. The second row shows the results without the meta-constraint for those solved instances.

	μ	σ	Time	Cost	Cost (TO)	Rel. diff.	#improved
Percent 6	9.39	1.10	89.47	234.72	263.06	5.26	1875
Normal	9.14	1.30	5.27	228.56	250.81	7.72	-

Table 6.4: Results when adding the homogeneousPercentWeight meta-constraint with factor 6. Statistics for the 2109 solved instances with a timeout of 600 seconds. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; Time: mean solving time (in seconds); Cost: mean optimal cost; Cost (TO): mean of best lower bounds for those instances that exceeded the timeout; Rel. diff.: mean of differences between relative percentual maximal and minimal costs; #improved: number of instances with improved relative difference.

The mean of the percent differences is reduced from 7.76 to 5.26, which is almost 32%. In particular, we reduce the percent difference between the most penalized nurse and the less penalized nurse in 1875 instances out of 2109. The average penalization per nurse increases from 9.14 to 9.39, just 2.74%, and the average global cost only increases from 228.56 to 234.72. However, the average solving time increases from 5.27 to 89.47 seconds for the solved instances. In fact, the solving time increases no doubt by much more than this on average if considering the timed-out instances.

As we did with the experiments for the absolute difference, we have conducted more experiments, in this case increasing the factor from 6 to 11. The results are re-

ported in Table 6.5. In this case, only 492 out of 5113 could not be solved within the timeout. The number of improved instances decreases but the solving time improves. Therefore, with the `homogeneousPercentWeight` meta-constraint we can also find a reasonable balance between the quality of the solutions and the required solving time with respect to the original problem.

	μ	σ	Time	Cost	Cost (TO)	Rel. diff.	#improved
Percent 11	9.62	1.67	33.28	240.38	269.14	8.42	1592
Normal	9.57	1.71	5.51	239.23	264.20	9.04	-

Table 6.5: Results when adding the homogeneousPercentWeight meta-constraint with factor 11. Statistics for the 4621 solved instances with a timeout of 600 seconds. μ: mean of means of costs of violated constraints per nurse; σ: standard deviation of means of costs of violated constraints per nurse; Time: mean solving time (in seconds); Cost: mean optimal cost; Cost (TO): mean of best lower bounds for those instances that exceeded the timeout; Rel. diff.: mean of differences between relative percentual maximal and minimal costs; #improved: number of instances with improved relative difference.

6.6.2 Soft Balanced Academic Curriculum Problem

The *Balanced Academic Curriculum Problem* (BACP) consists in assigning courses to academic periods satisfying prerequisite constraints between courses and balancing the workload (in terms of credits) and the number of courses of each period [CM01, HKW02]. In particular, given

- a set of courses, each of them with an associated number of credits representing the academic effort required to successfully follow it,
- a set of periods, with a minimum and maximum bound both on the number of courses and number of credits assigned to each period,
- and a set of prerequisites between courses stating that, if a course c has as prerequisite a course d, then d must be taught in a period previous to the one of c,

the goal of the BACP is to assign a period to every course satisfying the constraints on the bounds of credits and courses per period, and the prerequisites between courses. In the optimization version of the problem, the objective is to improve the balance of the workload (amount of credits) assigned to each period. This is achieved by minimizing the maximum workload of the periods.

There may be situations where the prerequisites make the instance unsatisfiable. We propose to deal with unsatisfiable instances of the decision version of the BACP by

relaxing the prerequisite constraints, i.e., by turning them into soft constraints. We allow the solution to violate a prerequisite constraint between two courses but then, in order to reduce the pedagogical impact of the violation, we introduce a new hard constraint, the corequisite constraint, enforcing both courses to be assigned to the same period. We call this new problem *Soft Balanced Academic Curriculum Problem* (SBACP).

The goal of the SBACP is to assign a period to every course minimizing the total amount of prerequisite constraint violations and satisfying the conditionally introduced corequisite constraints, and the constraints on the number of credits and courses per period.

In Figure 6.6 we propose a modelling of the SBACP using `WSimply`. In order to obtain instances of the SBACP, we have over-constrained the BACP instances from the MiniZinc [NSB+07] repository, by reducing to four the number of periods, and proportionally adapting the bounds on the workload and number of courses of each period. With this reduction on the number of periods, we have been able to turn into unsatisfiable all these instances.

In the following we present several experiments with `WSimply` over the obtained SBACP instances, using a 2.6GHz Intel® Core™ i5, with a timeout of 600 seconds.

The best solving approach for this problem in our system is yices, closely followed by dico. The core solving approach is not competitive for this problem. The performance of the WPM1 algorithm strongly depends on the quality of the unsatisfiable cores the SMT solver is able to return at every iteration. This quality has to do, among other details, with the size of the core, the smaller the better, and the overlapping of the cores, the lower the better. For the SBACP instances, the SMT solver tends to return cores which involve almost all the soft clauses, i.e., they are as big as possible and they completely overlap. This clearly degrades the performance of the WPM1 algorithm.

Columns two to five of Table 6.6 show the results obtained by `WSimply` on our 28 instances. The second column shows the required CPU time in seconds (with the yices solving approach); the third column indicates the total amount of prerequisite violations, and the fourth and fifth columns show the maximum and minimum number of prerequisite constraint violations per course. This maximum and minimum exhibit the lack of homogeneity of each instance. Column six shows the time obtained by CPLEX solving the SBACP instances.

As we can observe, there exist instances which have courses with three, four, and even five prerequisite constraint violations, as well as courses with zero violations. It could be more egalitarian to obtain solutions where the difference in the number of prerequisite constraint violations between courses is smaller. Thanks to the meta-constraint `homogeneousAbsoluteNumber` we can easily enforce this property of the solutions, as shown in Figure 6.7.

```
Problem:sbacp
 Data
    int n_courses;
    int n_periods;
    int load_per_period_lb;
    int load_per_period_ub;
    int courses_per_period_lb;
    int courses_per_period_ub;
    int course_load[n_courses];
    int n_prereqs;
    int prereqs[n_prereqs,2];
 Domains
    Dom dperiods=[1..n_periods];
    Dom dload=[load_per_period_lb..load_per_period_ub];
    Dom dcourses=[courses_per_period_lb..courses_per_period_ub];
 Variables
    IntVar course_period[n_courses]::dperiods;
    IntVar period_load[n_periods]::dload;
    IntVar period_courses[n_periods]::dcourses;
 Constraints
 %%% Hard Cosntraints
 %%%
 %%% Every course must be assigned to exactly one period.
 %%% This constraint is implicit in this modelling.
 %%%
 %%% The number of courses per period must be within bounds
 %%%  (i.e. the domain of period_courses[p]).
     Forall(p in [1..n_periods]) {
       Count([course_period[c]|c in [1..n_courses]],p,period_courses[p]);
     };
 %%% The workload in each period must be within bounds
 %%%  (i.e. the domain of period_load[p]).
     Forall(p in [1..n_periods]) {
       Sum([If_Then_Else(course_period[c]=p)(course_load[c])(0)
            |c in [1..n_courses]],period_load[p]);
     };
 %%% If a prerequisite is violated then the corequisite constraint is mandatory.
     Forall(np in [1..n_prereqs]) {
       Not(pre[np])
       Implies(course_period[prereqs[np,1]]=course_period[prereqs[np,2]]);
     };
 %%% Soft Constraints
 %%%
 %%% Prerequisites are desirable but non-mandatory.
 %%% Each prerequisite (np) is posted as a soft constraint with
 %%%  its label (pre[np]) and a violation cost of 1.
     Forall(np in [1..n_prereqs]) {
       #pre[np]:(course_period[prereqs[np,1]]>course_period[prereqs[np,2]])@1;
     };
```

Figure 6.6: WSimply model for the SBACP.

Also in Table 6.6, we show the results obtained by `WSimply` on these instances with homogeneity factor $\mathtt{F} = 1$ (second block of columns) and $\mathtt{F} = 2$ (third block of columns). The homogeneity factor bounds the difference in the violation of prerequisite constraints between courses (1 and 2 in our experiments). For homogeneity with factor 1, there are 5 unsolvable instances and 9 instances that achieve homogeneity by increasing

						Homogeneity factor 1				Homogeneity factor 2			
N.	Time	Cost	V. per c.		CPLEX	Time	Cost	V. per c.		Time	Cost	V. per c.	
			Max	Min				Max	Min			Max	Min
1	0.63	19	2	0	0.45	0.26	21	1	0	0.75	19	2	0
2	4.27	16	2	0	0.39	0.27	42	2	1	1.61	16	2	0
3	0.78	17	2	0	0.83	0.26	19	1	0	1.7	17	2	0
4	32.93	28	4	0	0.69	0.26	unsatisfiable			3.69	28	**2**	0
5	0.97	15	2	0	0.43	0.25	39	2	1	0.86	15	2	0
6	0.56	10	2	0	0.43	0.31	10	**1**	0	0.47	10	2	0
7	1.35	19	2	0	0.50	0.28	40	2	1	0.86	19	2	0
8	2.63	21	3	0	0.46	0.24	unsatisfiable			0.56	23	2	0
9	5.44	27	3	0	0.92	0.22	unsatisfiable			1.26	27	**2**	0
10	3.43	21	3	0	0.57	0.28	39	2	1	3.07	21	**2**	0
11	10.23	22	3	0	0.59	0.29	38	2	1	2.29	22	**2**	0
12	18.11	27	3	0	0.66	0.3	47	2	1	4.3	27	**2**	0
13	1.29	14	3	0	0.32	0.28	17	1	0	0.72	15	2	0
14	0.47	17	2	0	0.40	0.44	33	2	1	0.34	17	2	0
15	0.17	6	2	0	0.20	0.45	28	2	1	0.26	6	2	0
16	1.61	15	2	0	0.31	0.29	15	**1**	0	1.1	15	2	0
17	10.72	23	5	0	0.66	0.24	unsatisfiable			0.24	unsatisfiable		
18	2.93	20	3	0	0.54	0.23	unsatisfiable			1.09	20	**2**	0
19	0.43	16	2	0	0.37	0.25	39	2	1	0.41	16	2	0
20	3.71	15	2	0	0.59	0.49	15	**1**	0	3.58	15	2	0
21	1.93	14	2	0	0.47	0.25	20	1	0	0.61	14	2	0
22	0.74	15	2	0	0.43	0.31	17	1	0	0.55	15	2	0
23	2.18	20	1	0	0.63	0.28	20	1	0	2.33	20	1	0
24	0.22	7	2	0	0.30	0.32	9	1	0	0.3	7	2	0
25	3.03	13	2	0	0.33	0.52	14	1	0	1.58	13	2	0
26	0.23	5	1	0	0.21	0.38	5	1	0	0.35	5	1	0
27	1.09	17	2	0	0.43	0.25	21	1	0	1.48	17	2	0
28	0.19	10	2	0	0.28	0.26	11	1	0	0.34	10	2	0
T.	1.48	451			0.44	0.28	209			0.86	3		

Table 6.6: Results of the experiments on the SBACP instances without and with homogeneity. Numbers in boldface denote instance improvements maintaining the same cost. The last row shows the median of CPU solving time and the sum of the costs found; in the homogeneity cases we show the aggregated increment of the cost with respect to the original instances.

the minimum number of violations per course (from 0 to 1) with, in addition, a dramatic increase on the total number of violations (+209). Experiments with homogeneity factor 2 give different results on 9 instances, all of which, except one becoming unsatisfiable, are effectively improved by reducing the maximum number of violations per course, and slightly increasing the total number of violations (+3). Interestingly, the solving time has been improved when adding homogeneity.

```
 %%% We enforce homogeneity with factor F, over the lists of
 %%% soft constraints on prerequisites for each course.
homogeneousAbsoluteNumber([[pre[np]|np in [1..n_prereqs],
      prereqs[np,1]=c]|c in [1..n_courses]],F);
```

Figure 6.7: WSimply constraints to add to the model for the SBACP in order to ask for homogeneity with factor `F` in the solutions.

By way of guidance a comparison between `WSimply` and CPLEX has been done only over the basic SBACP instances since CPLEX doesn't have any meta-constraint. `WSimply` exhibits a reasonable good performance taking 1.48 seconds in median against 0.44 seconds of CPLEX.

	Homogeneity factor 2					MLevel: prereq, workload					MLevel: prereq, wl, courses				
N.		c./p.		wl/p.			c./p.		wl/p.			c./p.		wl/p.	
	Time	Min	Max	Min	Max	Time	Min	Max	Min	Max	Time	Min	Max	Min	Max
1	0.75	11	14	51	87	23.6	12	14	65	**66**	32.68	12	**13**	65	66
2	1.61	12	13	68	77	11.06	11	15	70	**71**	9.33	12	**13**	70	71
3	1.7	12	14	62	72	23.59	11	15	67	**68**	17.00	11	**14**	66	68
4	3.69	11	17	52	112	16.09	11	15	74	**77**	5.29	12	**13**	72	77
5	0.86	11	15	41	81	6.2	9	16	59	**63**	8.67	9	**15**	58	63
6	0.47	11	15	49	86	2.35	12	13	59	**60**	2.93	12	13	59	60
7	0.86	6	17	32	104	9.73	11	14	65	**66**	59.32	10	14	65	66
8	0.56	9	16	43	86	1.92	10	16	58	**66**	2.66	12	**13**	61	66
9	1.26	8	16	39	109	15.28	12	13	74	**77**	5.00	12	13	74	77
10	3.07	8	15	33	79	26	11	16	58	**66**	12.19	12	**14**	57	66
11	2.29	10	15	46	88	12.26	11	14	68	**68**	45.67	12	**13**	68	68
12	4.3	6	16	37	100	97.92	10	17	69	**70**	195.52	10	17	69	70
13	0.72	6	18	44	98	10.16	10	15	71	**72**	16.06	11	**13**	71	72
14	0.34	9	18	40	106	1.18	10	16	65	**69**	1.79	10	16	65	69
15	0.26	5	16	46	92	13.05	11	16	72	**72**	46.65	11	**13**	72	72
16	1.1	9	17	50	79	61.86	11	14	61	**63**	108.07	12	**13**	61	63
17	0.24		unsatisfiable			0.59		unsatisfiable			0.67		unsatisfiable		
18	1.09	10	15	62	92	15.25	12	13	74	**75**	19.96	12	13	74	75
19	0.41	8	19	51	99	8.65	11	14	67	**68**	15.94	11	**13**	67	68
20	3.58	10	16	54	112	53.86	10	14	70	**75**	38.8	10	14	70	75
21	0.61	9	18	42	94	2.27	11	14	65	**65**	2.51	12	**13**	65	65
22	0.55	10	16	57	105	2.18	11	14	75	**77**	1.76	12	**13**	75	77
23	2.33	20	25	25	128	103.47	11	15	67	**68**	173.60	11	**13**	67	68
24	0.3	12	14	57	81	0.71	12	13	63	**78**	2.65	12	13	64	78
25	1.58	9	16	44	87	31.86	12	14	70	**70**	26.15	12	**13**	70	70
26	0.35	5	18	24	102	9.28	11	14	51	**78**	8.01	10	14	61	78
27	1.48	10	15	57	96	8.19	11	15	81	**81**	23.60	11	**14**	81	81
28	0.34	9	15	42	101	2.3	11	15	69	**70**	4.28	11	**14**	68	70
T.	0.86		439		2553	10.61				-654	14.07		-27		

Table 6.7: Results when adding the multiLevel Adding the multiLevel meta-constraint to deal with the optimization version of the SBACP, minimizing the maximum workload per period and the maximum number of courses per period. Numbers in boldface denote improvements. Timeout is 600s. The last row shows the median of the solving time and the improvements on the aggregated maximums of the workload and number of courses per period thanks to the `multiLevel` meta-constraint.

The first block of columns of Table 6.7 show the minimum and maximum number of courses per period and the minimum and maximum workload per period, for each considered instance, when asking for homogeneity with factor 2 on the number of prerequisite violations.[7] As we can see, the obtained curricula are not balanced enough with respect to the number of courses per period and the workload per period. Therefore, we propose to consider the optimization version of SBACP by extending the initial modelling and using the `multiLevel` meta-constraint in order to improve the balancing in the workload and the number of courses per period (recall that the homogeneity meta-constraint on the number of violations introduced so far is hard). In Figure 6.8 we show how we have implemented this extension for the workload (for the number of courses it can be done analogously). We also must set to undefined the weights of the prerequisite soft constraints to let the `multiLevel` meta-constraint to compute them.

```
int load_distance=load_per_period_ub-load_per_period_lb;
%%% load_bound is the upper bound of the load of all periods.
IntVar load_bound::dload;
Forall(p in [1..n_periods]) {
 period_load[p]=<load_bound;
};
%%% We post as soft constraints each unit of distance
%%% between load_bound and load_per_period_lb.
Forall(d in [1..load_distance]) {
 #bala[d]:(load_bound<(load_per_period_lb + d))@{_};
 };
%%% We compose the new objective function with these two
%%% components, being more important the prerequisites
%%% than the minimization of the maximum load per period.
multiLevel([[pre[np]|np in [1..n_prereqs]],
     [bala[d]|din [1..load_distance]]]);
```

Figure 6.8: Extension to minimize the maximum workload (amount of credits) of periods.

The idea of this encoding is to minimize the maximum workload per period using soft constraints. Column eleven (wl/p. Max) shows the improvement on the maximum workload per period obtained when introducing its minimization with the `multiLevel` meta-constraint. Column fourteen (c./p. Max) shows the improvement on the the maximum number of courses per period obtained when adding its minimization as the next level in the `multiLevel` meta-constraint.

[7] We have chosen homogeneity factor 2 to continue with the experiments since, with factor 1, the number of violations increases in almost all instances.

6.7 Extensional WCSP

Most of the existing WCSP solvers are developed for solving extensional WCSPs where the constraints are specified by means of tuples of variables values denoting goods or no-goods. This is not the purpose of `WSimply`, designed to solve intensional WCSP. Nevertheless, for the sake of completeness we have implemented a system to encode extensional WCSPs into SMT. Thanks to this system we are able to compare the efficiency of SMT with some WCSP solvers.

Basically our system deals with tuples of goods using a support encoding. We are considering extensional WCSP instances described in XCSP.

Example 26 *Let us consider the following relation $R2$ extensionally describing a soft constraint with 6 no-goods tuples (in XCSP format):*

```
<relation name="R2" arity="2" nbTuples="6" semantics="soft"
    defaultCost="0">
      5:0 0|0 3|1 1|2 2|3 0|3 3
    </relation>
```

The encoding into WSMT, used in our system, of the constraint resulting of applying $R2$ on the two variables X_0 and X_1 with domain $\{0 \dots 3\}$, is the following:

$$\textit{domain encoding} \quad \begin{cases} (X_0 \geq 0, \infty) \\ (X_0 \leq 3, \infty) \\ (X_1 \geq 0, \infty) \\ (X_1 \leq 3, \infty) \end{cases}$$

$$\textit{goods encoding} \quad \begin{cases} (X_0 \neq 0 \vee X_1 = 1 \vee X_1 = 2, 5) \\ (X_0 \neq 1 \vee X_1 = 0 \vee X_1 = 2 \vee X_1 = 3, 5) \\ (X_0 \neq 2 \vee X_1 = 0 \vee X_1 = 1 \vee X_1 = 3, 5) \\ (X_0 \neq 3 \vee X_1 = 1 \vee X_1 = 2, 5) \end{cases}$$

As solving mechanism, we have used the same three approaches of `WSimply`. We have run our experiments on machines with the following specifications. Operating system: Rocks Cluster 5.2 Linux 2.6.18; processor: AMD Opteron 242 Processor 1.5 GHz; memory: 450 MB; cache: 1024 KB. The instances that we have used are from the WCSP benchmarks of the `http://www.cril.univ-artois.fr/~lecoutre/benchmarks.html` repository. The instances are in XCSP 2.1 format (see [RL09]).

Table 6.8 presents the percentage of solved instances for each of the 17 families of instances with a timeout of 1 hour. toulbar [DGSA] is the best performing solver. However our SMT approaches perform reasonably well. Overall, the average percentage of instances solved by toulbar is 87%, while for the whole SMT approach is 79%. For the *celar* and *pedigree* family we found that the extensional representation in SMT consumed too much memory according to the restrictions of our system.

Instance set	#	toulbar	yices	dico	core
academics	15	**93**	**93**	86	**93**
bwt	10	**100**	**100**	**100**	**100**
celar	15	**93**	0	0	0
coloring	22	**77**	45	40	63
depot	4	**100**	**100**	**100**	**100**
dimacs	25	92	**100**	**100**	**100**
driver	22	**100**	**100**	**100**	72
graphs	10	**30**	20	0	10
jnh	44	**100**	**100**	**100**	79
logistics	4	**100**	**100**	**100**	**100**
mprime	13	**92**	**92**	**92**	**92**
pedigree	19	**100**	73	63	73
rover	4	**100**	**100**	**100**	**100**
satellite	7	**100**	**100**	**100**	57
spot5	21	23	14	19	**28**
warehouse	55	**85**	0	0	67
zenotravel	8	**100**	**100**	**100**	**100**

Table 6.8: Comparison between WSMT and Toulbar, showing the number of instances and the percentage of solved instances (timeout of 1h).

6.8 Summary

We have introduced a new framework, called `WSimply`, which fills the gap between CSP and SMT regarding over-constrained problems. A new modelling language has been introduced for the intensional description of over-constrained problems. Our system supports some of the best-known meta-constraints from the literature and news ones. Meta-constraints increase the capability to easily model several real-world problems.

The usage of SMT solvers in our solving strategies is a promising choice, since several constraints, once described intensionally, can be potentially more efficiently handled. We have incorporated adaptations of UNSAT core based algorithms from the MaxSAT

community to our system.

We have provided the results of some experimentation showing the good performance of solving WCSP with SMT and finally we have described an encoding of extensional WCSPs into SMT.

Chapter 7

Scheduling

In Chapters 5 and 6 we have seen that SMT is a good approximation for solving CSP, COP and WCSP. It is competitive with the best approaches (finite domain constraint programming systems, MILP, SAT, lazy clause generation, ...). Its performance is really good in almost all types of problems, but especially in problems with a strong Boolean component with integer arithmetic. Therefore, SMT is very well suited for scheduling problems. In scheduling problems there is an important integer arithmetic component: precedence delays between activities, sums to verify that activities running at a certain time do not exceed an specified resource capacity, etc. Likewise, incompatibilities and precedences between activities, as well as many other details, can be encoded with Boolean formulas (possibly using arithmetic predicates as atoms). Another factor that makes this type of problem very suitable for SMT is that, when searching for a solution, many conflicts arise, and hence many things can be learned in the form of lemmas: a certain activity must necessarily go after another (due to a combination of precedence requirements and resource consumptions), etc.

In this chapter we show that SMT is not only competitive with generic CP tools, but also that algorithms built on top of an SMT solver can have equal or better performance than other ad hoc programs based on other approaches and designed specifically for a given problem. As an example we have chosen the resource-constrained project scheduling problem (RCPSP), which is the scheduling problem more widely discussed in the literature. We show that adequate encodings and algorithms built on top of an SMT solver can result in an extraordinary approach.

Since the RCPSP is an optimization problem there exist at least two natural approaches for solving it with SMT: on the one hand, we can simply iteratively encode the RCPSP instance into successive SMT instances that bound the optimal makespan; on the other hand, we can encode the RCPSP into a Weighted SMT instance where the soft clauses encode the objective function.

We have built a tool, named `rcpsp2smt`,[1] which is able to solve RCPSP instances using the Yices (Weighted)SMT solver [DdM06b]. Apart from using the Yices default algorithm for solving the Weighted SMT instances we have implemented, through the Yices API, an algorithm based on unsatisfiable cores [ABL09, MSP09] presented in Subsection 6.5.

We first briefly review the state-of-the-art in RCPSP solving (Section 7.1) to then formally introduce the problem and its notation (Section 7.2). Next we describe our tool `rcpsp2smt` with special emphasis in the preprocessing (Section 7.3) and optimization phases (Section 7.4). In Section 7.5 we describe the four different types of encodings that we have implemented, with a large number of variants for each, as well as many formulations of propagators and redundancy constraints. For each of the encodings we widely discuss the advantages and disadvantages and try to reason why they improve or worsen the efficiency of the SMT solver. In Section 7.5.5 we present a entirely new event-based encoding for the RCPSP. Finally, we present an extensive interpretation of the results obtained (Section 7.6) and include a summary of the contributions presented in this chapter (Section 7.7).

The content of this chapter is part of our publication *Satisfiability modulo theories: An efficient approach for the resource-constrained project scheduling problem* [ABP+11b].

This chapter corresponds to the fourth objective of the thesis (to provide an SMT based system being competitive with state-of-the-art methods for scheduling problems) and provides the eighth contribution.

7.1 State-of-the-Art in the RCPSP

The Resource Constrained Project Scheduling Problem (RCPSP) consists in scheduling a set of non-preemptive activities with predefined durations and demands on each of a set of renewable resources, subject to partial precedence constraints. Normally the goal is to minimize the makespan. This is one of the most general scheduling problems that has been extensively studied in the literature. Some surveys published in the last years include [HRD98, BDM+99, KP01, HL05] and, more recently, [HB10, KALM11]. The RCPSP is NP-hard in the strong sense [BMR88, BLK83]. However, many small instances (with up to 50 activities) are usually tractable within a reasonable time.

Many approaches have been considered in order to solve the RCPSP: constraint programming (CP) [LM08, Bap09], Boolean satisfiability (SAT) [Hor10], mixed integer linear programming (MILP) [KALM11], branch and bound algorithms (BB) [DPPH00] and others [DV07]. A hybrid approach using SAT and CP lazy clause generation technol-

[1] `rcpsp2smt` is available at `http://ima.udg.edu/recerca/lap/rcpsp2smt/`

ogy [SFSW09, SFSW10] (see Section 4.6) has shown very good results.

In the literature there are no RCPSP solvers based on SMT. This is the first attempt. The only comparable approach is the new Lazy_fd which, as mentioned in Section 4.6, uses a SAT solver together with a finite domain solver, being the best known approach for the RCPSP. We compare our results with the results of that approach.

We use instances from the web page `http://129.187.106.231/psplib`. This page contains a library of different instance sets, as well as optimal and heuristic solutions, for various types of resource constrained project scheduling problems. We also compare our results with the best published results on this page.

7.2 The Resource-Constrained Project Scheduling Problem

The RCPSP[2] is defined by a tuple (V, p, E, R, B, b) where:

- $V = \{A_0, A_1, \ldots, A_n, A_{n+1}\}$ is a set of activities. A_0 and A_{n+1} are dummy activities representing by convention, the starting and the finishing activities respectively. The set of non-dummy activities is defined by $A = \{A_1, \ldots, A_n\}$.
- $p \in \mathbb{N}^{n+2}$ is a vector of durations. p_i denotes the duration of activity i, with $p_0 = p_{n+1} = 0$ and $p_i > 0, \forall i \in \{1, \ldots, n\}$.
- E is a set of pairs representing precedence relations, thus $(A_i, A_j) \in E$ means that the execution of activity A_i must precede that of activity A_j, i.e., activity A_j must start after activity A_i has finished. We assume that we are given a precedence activity-on-node graph $G(V, E)$ that contains no cycles; otherwise the precedence relation is inconsistent. Since the precedence is a transitive binary relation, the existence of a path in G from the node i to node j means that activity i must precede activity j. We assume that E is such that A_0 is a predecessor of all other activities and A_{n+1} is a successor of all other activities.
- $R = \{R_1, \ldots, R_m\}$ is a set of m renewable resources.
- $B \in \mathbb{N}^m$ is a vector of resource availabilities. B_k denotes the available amount of each resource R_k.
- $b \in \mathbb{N}^{(n+2)\times m}$ is a matrix of the demands of the activities for resources. $b_{i,k}$ represents the amount of resource R_k used during the execution of A_i. Note that $b_{0,k} = 0$, $b_{n+1,k} = 0$ and $b_{i,k} \geq 0, \forall i \in \{1, \ldots, n\}, \forall k \in \{1, \ldots, m\}$.

[2]This problem is denoted as $PS|prec|C_{max}$ in [BDM+99] and $m, 1|cpm|C_{max}$ in [HL05].

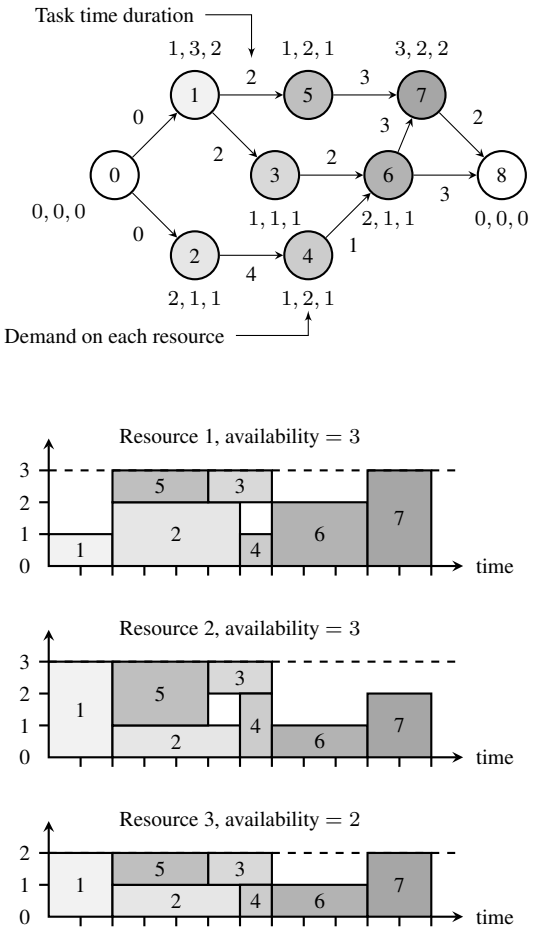


Figure 7.1: An example of RCPSP [LM08]

A schedule is a vector $S = (S_0, S_1, \ldots, S_n, S_{n+1})$ where S_i denotes the start time of each activity $A_i \in V$. We assume that $S_0 = 0$. A solution of the RCPSP problem is a non-preemptive (an activity cannot be interrupted once it is started) schedule S of minimal makespan S_{n+1} subject to the precedence and resource constraints:

$$\text{minimize } S_{n+1} \tag{7.1}$$

subject to:

$$S_j - S_i \geq p_i \qquad \forall (A_i, A_j) \in E \tag{7.2}$$

$$\sum_{A_i \in \mathcal{A}_t} b_{i,k} \leq B_k \qquad \forall B_k \in B, \forall t \in H \tag{7.3}$$

A schedule S is feasible if it satisfies the generalized precedence constraints (7.2) and the resource constraints (7.3) where $\mathcal{A}_t = \{A_i \in A \mid S_i \leq t < S_i + p_i\}$ represents the set

of non-dummy activities in process at time t, the set $H = \{0, \ldots, T\}$ is the scheduling horizon, and T (the length of the scheduling horizon) is an upper bound for the makespan.

In the example of Figure 7.1, three resources and seven (non-dummy) activities are considered. Each node is labeled with the number of the activity it represents. The durations of the activities are indicated in the on-going arcs, and the resource consumptions are indicated in the labels next to the nodes. The upper part of the picture represents therefore the instance to be solved, while the bottom part gives a feasible solution using Gantt charts. For each resource, the horizontal axis represents the time and the vertical axis represents the consumption.

In the following we describe the tool that we have developed, named `rcpsp2smt`, to solve the RCPSP using an SMT solver.

7.3 Preprocessing

The input of `rcpsp2smt` is an RCPSP instance in rcp or sch format,[3] which is preprocessed in order to build a more suitable instance for our solving method. After the preprocessing phase, `rcpsp2smt` searches for an optimal solution as described in Section 7.4.

In the preprocessing phase we can compute an extended precedence set, lower and upper bounds for the makespan,[4] time windows for each activity and a matrix of incompatibilities between activities. The suitability of each preprocess calculation for each encoding is discussed in Section 7.6.

Extended Precedence Set

Since a precedence is a transitive relation, we can compute the minimum precedence between each pair of activities in E. For this calculation we use the $\mathcal{O}(n^3)$ Floyd-Warshall algorithm on the graph defined by the precedence relation E labeling each arc (A_i, A_j) with the duration p_i. This extended precedence set is named E^* and contains, for each pair of activities A_i and A_j such that A_i precedes A_j, a tuple of the form $(A_i, A_j, l_{i,j})$ where $l_{i,j}$ is the length of the longest path from A_i to A_j. Note also that, if $(A_i, A_i, l_{i,i}) \in E^*$ for some A_i and $l_{i,i} > 0$, then there is a cycle in the precedence relation and therefore the problem is inconsistent and has no solution.

[3] RCPSP formats from PSPLib [KS97].

[4] End time of the schedule.

Lower Bound

A lower bound LB of the makespan is a known time at which the last activity A_{n+1} cannot start before. There are different methods for computing lower bounds (see [KS99, MMRB98]). We have implemented two of them:

- $LB1$*: Critical path bound.* This is the most obvious lower bound. To compute it, we ignore the capacity restrictions. Then, the minimal project duration is the length of the critical path in the project network. The critical path is the longest path between the initial activity A_0 and the final activity A_{n+1} in the graph of precedences. For instance, in Figure 7.1 the critical path is $[A_0, A_2, A_4, A_6, A_7, A_8]$ and has length 10. Note that we can easily know the length of this path if we have already computed the extended precedence set since we only need to obtain $l_{0,n+1}$ from $(A_0, A_{n+1}, l_{0,n+1}) \in E^*$.

- $LB2$*: Capacity bound.* To compute this bound, we ignore the precedence restrictions. Its value is the maximum of the division of the total requirement for each resource, by the capacity of the resource (rounded up to the next integer). The additional cost to compute this lower bound is $\mathcal{O}(nm)$ (being n the number of activities and m the number of resources). For instance, in the example of Figure 7.1 the capacity bound of resource 3 is 11.

$$LB2 = \max\{\lceil(\sum_{A_i \in A} b_{i,k} * p_i)/B_k\rceil \mid B_k \in B\}$$

  We set our computed lower bound to $LB = \max\{LB1, LB2\}$.

Upper Bound

An upper bound UB of the makespan is a known time at which the last activity A_{n+1} can start. We have considered two possibilities for the UB.

- The *trivial upper bound* is simply the sum of the duration of all the activities:

$$UB = \sum_{A_i \in A} p_i$$

- The *heuristic upper bound* method consists in using a fast heuristic method to find a (presumably not optimal) solution and using its makespan as the upper-bound. The heuristic that we have used is the parallel scheduling generation scheme (parallel SGS) algorithm proposed in [BB82] and described in [Kol96, HK00, KH06].

The parallel method for n activities consists of at most n stages in each of which a set of activities is scheduled. Each stage s is associated with a schedule time t_s (where $t_m \leq t_s$ for $m \leq s$). There are three activity sets:

- Complete set C: activities scheduled, which are completed up at the schedule time t_s.
- Active set A: activities scheduled, but which are at the schedule time t_s still active.
- Decision set D: activities not scheduled which are available for scheduling w.r.t. precedence and resource constraints.

Each stage consists of two steps:

- Determine the new t_s: the earliest completion time of activities in the active set A. The activities with a finish time equal to the new t_s are removed from A and put into C.
- One activity from D is selected with a priority rule (in our case the activity with smallest label) and scheduled to start at the current schedule time. The set D is recalculated. This step is repeated until D is empty.

The method terminates when all activities are scheduled.

For instance, in the example of Figure 7.1, the trivial upper bound is 22 and the one obtained with the heuristic method is 13.

Time Windows

We can reduce the domain of each variable $S_i \in \{S_1, \ldots, S_n, S_{n+1}\}$ (start time of activity A_i), that initially is $\{0\,..\,UB - p_i\}$, by computing its time window. The time window of activity A_i is $[ES_i, LS_i]$, being ES_i the earliest start time and LS_i the latest start time. To compute the time window we use the lower and upper bound and the extended precedence set as follows:

For activities $A_i, 1 \leq i \leq n$,

$$\begin{array}{rcll} ES_i & = & l_{0,i} & (A_0, A_i, l_{0,i}) \in E^* \\ LS_i & = & UB - l_{i,n+1} & (A_i, A_{n+1}, l_{i,n+1}) \in E^* \end{array}$$

and, for activity A_{n+1},

$$ES_{n+1} = LB \qquad LS_{n+1} = UB$$

For instance, in the example of Figure 7.1, activity A_4 has time window $[4, 16]$.

Note that, if E^* has been successfully computed, then $l_{0,i} \geq 0$ for all $(A_0, A_i, l_{0,i}) \in E^*$ and $l_{i,n+1} \geq p_i$ for all $(A_i, A_{n+1}, l_{i,n+1}) \in E^*$.

Incompatibility

We compute a matrix of Booleans I, where each element $I[i,j]$ (noted as $I_{i,j}$) is true if the activity A_i and the activity A_j cannot overlap in time. The incompatibility can occur for two reasons:

- *Precedence.* There exists a precedence constraint between A_i and A_j, that is, $(A_i, A_j, l_{i,j}) \in E^*$ or $(A_j, A_i, l_{j,i}) \in E^*$.
- *Resources.* For some resource R_k, the sum of the demands of the two activities A_i and A_j is greater than the resource capacity B_k: $\exists R_k \in R$ s.t. $b_{i,k} + b_{j,k} > B_k$.

For instance, in the example of Figure 7.1, activity A_4 is incompatible with activities A_0, A_2, A_6, A_7 and A_8 due to the precedences, and with activities A_1, A_5 and A_7 due to resource demands.

This matrix of incompatibilities is symmetric, and the additional cost for its computation is $\mathcal{O}(n^2 m)$.

Note that we could compute incompatibilities due to resource demands between subsets of activities in general (i.e., not restricted to only two activities). We have explored this approximation for subsets of size bigger than two but the gain in efficiency has been almost zero or even negative, in all encodings and solving methods.

7.4 Solving

We follow two approaches to solve the RCPSP with SMT solvers, similarly as we do in Subsection 6.5.5. Both approaches share the constraints derived from the preprocessing steps, and the ones from Equation (7.2) and (7.3) modelled as Section 7.5 describes. The difference consists on how we treat the objective function.

On the one hand, we implement an ad hoc search procedure which calls the SMT solver successively constraining the domain of the variable S_{n+1}, by adding a constraint $S_{n+1} \leq bound$ (where $bound$ is a constant such that $LB \leq bound \leq UB$). The value of $bound$ at each call to the solver depends on the chosen strategy. In fact, this procedure is almost the same than Algorithm 14 of Subsection 5.4.4. In our experiments we have considered the following bounding strategies:

- dichotomic (dico): the value of $bound$ is set to $\lfloor (LB + UB)/2 \rfloor$. If, after adding the constraint $S_{n+1} \leq bound$, the solver is able to find a model, UB is set to S_{n+1} (note that when we get a satisfiable answer, we can eventually improve our upper bound since the value of S_{n+1} can be strictly smaller than $bound$); otherwise LB is set to

$bound + 1$. Then, we compute the new $bound$ for the next call with the updated LB or UB. The process ends when UB and LB converge to the same value.

- linear-down (linear): the value of $bound$ is set to $UB - 1$. If, after adding the constraint $S_{n+1} \leq bound$, the solver is able to find a model, UB is set to S_{n+1} and we compute the new $bound$ for the next call with the updated UB; otherwise the process ends.

- hybrid (hybrid): this strategy consists in following the dichotomic one until a certain (parameterizable) threshold on the difference of UB and LB is reached; then the system shifts to the linear-down strategy.

On the other hand, some SMT solvers, like Yices, can solve Weighted SMT instances. Weighted SMT allows us to represent optimization problems. We just need to transform the objective function into a set of soft constraints, and keep the rest of the constraints as hard. In order to translate the objective function, we can simply enumerate all its possible values. For example, taking into account the RCPSP in Figure 7.1, where $LB = 11$ and $UB = 22$, we add the following set of soft constraints: $\{(S_{n+1} \leq 11, 1), (S_{n+1} \leq 12, 1), \ldots, (S_{n+1} \leq 22, 1)\}$. Each soft constraint is represented by a pair (C, w) where C is a constraint and w the weight (or cost) of falsifying C. We can obtain a more compact encoding by considering the binary representation of S_{n+1}. Following our example, we would add a new hard constraint, $1 \cdot b_0 + 2 \cdot b_1 + 4 \cdot b_2 + 8 \cdot b_3 + LB = S_{n+1}$, where b_i are integer variables, a new set of hard constraints $\{b_i \geq 0, b_i \leq 1 \mid 0 \leq i \leq 3\}$ and the set of soft constraints $\{(b_0 = 0, 1), (b_1 = 0, 2), (b_2 = 0, 4), (b_3 = 0, 8)\}$. The resulting instance can be solved by any SMT solver supporting Weighted SMT like, as said, Yices. However, since this is yet an immature research topic in SMT, we have extended the Yices framework by incorporating Weighted MaxSAT algorithms. In particular, we have implemented algorithms based on the detection of unsatisfiable cores. The performance of these algorithms heavily depends on the quality of the cores.

In our experiments we have considered the following Weighted SMT solving methods:

- Yices (yices): this method is the one directly provided by Yices for Weighted SMT instances.

- Weighted PM1 (wpm1): this method is an adaptation of the satisfiability test based algorithm for MaxSAT described in [ABL09], using the enumeration of the possible values of the objective function (see Subsection 6.5.5 and Algorithm 15).

- Weighted PM1 binary (wbo): this method is an adaptation of the satisfiability test based algorithm for MaxSAT described in [MSP09], using the binary representation of the objective function.

7.5 Encodings

SAT modulo linear integer arithmetic allows us to directly express all the constraints of the following encodings, since we can logically combine arithmetic predicates.[5] The four encodings we propose are inspired by existing ones, but conveniently adapted to SMT. Moreover, some refinements are introduced considering time windows, incompatibilities, and extended precedences. We also add redundant constraints for better propagation. Finally, we have completely reformulated the event-based modelling (see Subsection 7.5.5), obtaining much better performance than with the original one (see Subsection 7.5.4).

Since a schedule is a vector $S = (S_0, S_1, \ldots, S_n, S_{n+1})$ where S_i denotes the start time of each activity $A_i \in V$, in all encodings we use a set $\{S_0, S_1, \ldots, S_n, S_{n+1}\}$ of integer variables. By S' we denote the set $\{S_1, \ldots, S_n\}$.

Also, in all encodings the objective function is (7.1), and we have the following constraints:

$$S_0 = 0 \tag{7.4}$$

$$S_i \geq ES_i \qquad \forall A_i \in \{A_1, \ldots, A_{n+1}\} \tag{7.5}$$

$$S_i \leq LS_i \qquad \forall A_i \in \{A_1, \ldots, A_{n+1}\} \tag{7.6}$$

$$S_j - S_i \geq l_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{7.7}$$

where (7.5) and (7.6) are simple encodings of the time windows for each activity, and (7.7) encodes the extended precedences. Hence, in each encoding, besides their own there are $1 + 2(n+1) + |E^*|$ constraints.

We also introduce additional constraints for the incompatibilities between activities due to resource capacity constraints:

$$S_i + p_i \leq S_j \vee S_j + p_j \leq S_i \qquad \forall I_{i,j} \in I \text{ s.t. } I_{i,j} = true, \; (A_i, A_j, l_{i,j}) \notin E^* \text{ and } (A_j, A_i, l_{j,i}) \notin E^* \tag{7.8}$$

Notice that the incompatibilities between activities due to precedence constraints are already encoded by Equation (7.7).

7.5.1 Time Formulation

The most natural encoding for the RCPSP in SMT is the Time formulation, very similar to the MILP encoding proposed in [PW96] and referred in [KALM11] as Basic discrete-time formulation and in [SFSW09, SFSW10] as Time-resource decomposition.

[5] Precedence constraints can be expressed in the theory of difference logic, but the restrictions on resources can only be expressed in linear integer arithmetic.

The idea is that, for every time unit t and resource R_k, the sum of resource requirements of activities must be less than or equal to the resource availability.

$$ite((S_i \leq t) \wedge \neg(S_i \leq t - p_i);\ x_{i,t} = 1;\ x_{i,t} = 0) \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\} \tag{7.9}$$

$$\sum_{A_i \in A} ite(t \in \{ES_i, \ldots, LS_i + p_i - 1\};\ b_{i,k} * x_{i,t};\ 0) \leq B_k \quad \forall B_k \in B, \forall t \in H \tag{7.10}$$

where $ite(c;\ e_1;\ e_2)$ is an *if-then-else* expression denoting e_1 if c is true and e_2 otherwise. This is a typical SMT expression. Note that the condition $t \in \{ES_i, \ldots, LS_i + p_i - 1\}$ can be trivially encoded into $ES_i \leq t \wedge t \leq LS_i + p_i - 1$.

Constraints (7.9) impose that $x_{i,t} = 1$ if activity A_i is active at time t, and $x_{i,t} = 0$ otherwise. Note also that we are restricting the possible time units, and the integer variables, by using the time windows. Constraints (7.10) are a reformulation of the resource constraints (7.3) using the $x_{i,t}$ variables.

In this encoding we are adding $\sum_{i \in 1 \ldots n}(LS_i + p_i - ES_i)$ new integer variables and $\sum_{i \in 1 \ldots n}(LS_i + p_i - ES_i) + (T + 1) \cdot m$ new constraints to the fixed ones of (7.4) to (7.8), i.e., the number of new variables is $\mathcal{O}(nT)$ and the number of new constraints is $\mathcal{O}(T(n + m))$.[6]

Constraints (7.9) can be replaced with the following, by introducing new $\mathcal{O}(nT)$ Boolean variables $y_{i,t}$:

$$y_{i,t} \leftrightarrow (S_i \leq t) \wedge \neg(S_i \leq t - p_i) \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\} \tag{7.11}$$

$$ite(y_{i,t};\ x_{i,t} = 1;\ x_{i,t} = 0) \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\} \tag{7.12}$$

The addition of these new variables $y_{i,t}$ allows us to consider additional constraints to improve propagation:

$$(S_i = t) \rightarrow y_{i,t'} \quad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\}, \forall t' \in \{t, \ldots, t + p_i - 1\} \tag{7.13}$$

[6]Recall that T is the length of the scheduling horizon, n is the number of activities and m is the number of resources.

$$(S_i = t) \rightarrow \neg y_{i,t'} \qquad \begin{array}{l} \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\}, \\ \forall t' \in \{ES_i, \ldots, LS_i + p_i - 1\} \setminus \{t, \ldots, t + p_i - 1\} \end{array} \tag{7.14}$$

We have observed that for small problem instances this formulation gives better performance results than the previous. However, for big instances (with more than 50 activities) the addition of the new variables $y_{i,t}$ usually makes the problem intractable by our system.

The following redundant constraints help improving the search time in all instances. This is probably due to the fact that they are unit clauses, and hence they are restricting the domain of $x_{i,t}$ from the beginning:

$$x_{i,t} \geq 0 \qquad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\} \tag{7.15}$$

$$x_{i,t} \leq 1 \qquad \forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + p_i - 1\} \tag{7.16}$$

7.5.2 Task Formulation

In this formulation we use variables indexed by activities instead of by time. The key idea is that checking only that there is no overload at the beginning (end) of each activity is sufficient to ensure that there is no overload at every time unit (in the non-preemptive case). In this formulation, the number of variables and constraints is independent of the length T of the scheduling horizon. This formulation is similar to the Task-resource decomposition of and it is inspired by the encoding proposed in [OEK99] for temporal and resource reasoning in planning.

The constraints are the following, where the Boolean variables $z^1_{i,j}$ denote if activity A_i starts not after A_j. The Boolean variables $z^2_{i,j}$ denote if activity A_j starts before A_i ends. The integer variables $z_{i,j}$ are 1 if activity A_i is active when activity A_j starts, and 0 otherwise.

$$z^1_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{7.17}$$

$$\neg z^1_{j,i} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{7.18}$$

$$z^1_{i,j} \leftrightarrow S_i \leq S_j \qquad \begin{array}{l} \forall A_i, A_j \in A, \\ (A_i, A_j, l_{i,j}) \notin E^*, (A_j, A_i, l_{j,i}) \notin E^*, \\ i \neq j \end{array} \tag{7.19}$$

$$\neg z^2_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{7.20}$$

$$z^2_{j,i} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{7.21}$$

$$z^2_{i,j} \leftrightarrow S_j < S_i + p_i \qquad \forall A_i, A_j \in A, (A_i, A_j, l_{i,j}) \notin E^*, (A_j, A_i, l_{j,i}) \notin E^*, i \neq j \tag{7.22}$$

$$z_{i,j} = 0 \qquad \forall I_{i,j} \in I \text{ s.t. } I_{i,j} = true \tag{7.23}$$

$$ite(z^1_{i,j} \wedge z^2_{i,j};\ z_{i,j} = 1;\ z_{i,j} = 0) \qquad \forall I_{i,j} \in I \text{ s.t. } I_{i,j} = false, i \neq j \tag{7.24}$$

$$\sum_{A_i \in A \setminus \{A_j\}} b_{i,k} * z_{i,j} \leq B_k - b_{j,k} \qquad \forall A_j \in A, \forall B_k \in B \tag{7.25}$$

The last constraints (7.25) state that, for every activity A_j and resource R_k, the sum of the resource demands $b_{i,k}$ for R_k, from the activities A_i that overlap with A_j at its start time, should not exceed the capacity B_k of R_k less the demand $b_{j,k}$ for R_k from A_j.

This encoding involves $2n(n-1)$ Boolean variables, $n(n-1)$ integer variables and $3n(n-1) + nm$ (non unary)[7] constraints, i.e., the number of variables to add –to the ones present in the fixed constraints from (7.4) to (7.8)– is $\mathcal{O}(n^2)$ and the number of constraints to add is $\mathcal{O}(n^2 + nm)$. Hence, this formulation is not sensitive to the length T of the scheduling horizon.

If we use end time variables $E_i = S_i + p_i$, we can generate a symmetric model to the one defined above.

In order to improve propagation, we have the following redundant constraints encoding anti-symmetry and transitivity of the precedence relation:

$$z^1_{i,j} \vee z^1_{j,i} \qquad \forall A_i, A_j \in A, i \neq j \tag{7.26}$$

$$(z^1_{i,j} \wedge z^1_{j,k}) \rightarrow z^1_{i,k} \qquad \forall A_i, A_j, A_k \in A, i \neq j, j \neq k, i \neq k \tag{7.27}$$

The following redundant constraints have shown to slightly improve propagation in this encoding:

$$z^1_{i,j} \vee z^2_{i,j} \qquad \forall A_i, A_j \in A, i \neq j \tag{7.28}$$

$$z^2_{i,j} \vee z^2_{j,i} \qquad \forall A_i, A_j \in A, i \neq j \tag{7.29}$$

$$\neg z^1_{i,j} \vee z^2_{j,i} \qquad \forall A_i, A_j \in A, i \neq j \tag{7.30}$$

[7]The simple unary constraints (7.17), (7.18), (7.20), (7.21) and (7.23) (and similar constraints in other formulations as well) are not posted by our system. Instead, every occurrence of their variables is replaced by its (known) value in all expressions.

7.5.3 Flow Formulation

This formulation is inspired by the formulations of [AMR03, KALM11] named Flow-based continuous-time formulation. It models the flow of resources between finishing activities and starting activities. Once an activity finishes it transfers to some forthcoming starting activities the amount of resource it was using, and one activity must receive enough resources, according to its requirements, from finishing activities.

The Boolean variables $y_{i,j}$ denote if activity A_j starts after the completion of A_i. The integer variables $f_{i,j,k}$ denote the quantity of resource R_k that is transferred from A_i (when finished) to A_j (at the start of its processing). Note that in this formulation we are considering the whole set of activities $V = \{A_0, A_1, \ldots, A_n, A_{n+1}\}$, i.e., we include the dummy activities A_0 and A_{n+1}.

$$y_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{7.31}$$

$$\neg y_{j,i} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{7.32}$$

$$y_{i,j} \leftrightarrow S_j \geq S_i + p_i \qquad \begin{array}{l} \forall A_i, A_j \in V, \\ (A_i, A_j, l_{i,j}) \notin E^*, (A_j, A_i, l_{j,i}) \notin E^*, \\ i \neq j \end{array} \tag{7.33}$$

$$f_{i,j,k} \geq 0 \qquad \forall A_i, A_j \in V, R_k \in R, i \neq j \tag{7.34}$$

$$f_{i,j,k} \leq \min(b^e_{i,k}, b^e_{j,k}) \qquad \forall A_i, A_j \in V, R_k \in R, i \neq j \tag{7.35}$$

$$y_{i,j} \vee (f_{i,j,k} = 0) \qquad \forall A_i, A_j \in V, R_k \in R, i \neq j \tag{7.36}$$

$$\sum_{A_j \in A \cup \{A_{n+1}\}} f_{i,j,k} = b_{i,k} \qquad \forall A_i \in A \cup \{A_0\}, \forall R_k \in R, i \neq j \tag{7.37}$$

$$\sum_{A_i \in A \cup \{A_0\}} f_{i,j,k} = b_{j,k} \qquad \forall A_j \in A \cup \{A_{n+1}\}, \forall R_k \in R, i \neq j \tag{7.38}$$

Constraints (7.34) and (7.35) fix the range of variables $f_{i,j,k}$. In (7.35), $b^e_{q,k}$ denotes $b_{q,k}$ for all q such that $A_q \in A$, and B_k for $q = 0$ and $q = n + 1$. Constraints (7.36) state that, if A_j does not start after the completion of A_i, then the flow from activity A_i to activity A_j must be 0 for all resources R_k. Constraints (7.37) and (7.38) state that, for each activity and resource, the sum of the flows transferred and the sum of the flows received by the activity must be equal to its demand for the resource. Recall that $b_{0,k} = 0$ and $b_{n+1,k} = 0$ for all k. Hence, no flow enters activity A_0 and no flow exits activity A_{n+1}.

The constraints (7.8) for the incompatibilities are reformulated as follows:

$$y_{i,j} \vee y_{j,i} \qquad \forall I_{i,j} \in I \text{ s.t. } I_{i,j} = true, \; (A_i, A_j, l_{i,j}) \notin E^*, (A_j, A_i, l_{j,i}) \notin E^* \tag{7.39}$$

This encoding involves $(n+2)(n+1)$ Boolean variables, $(n+2)(n+1)m$ integer variables, and $(n+2)(n+1) + 3(n+2)(n+1)m + 2(n+1)nm + 2(n+1)m$ (non trivial) constraints, i.e., the number of variables and constraints to add –to the ones already in (7.4) to (7.7)– is $\mathcal{O}(n^2 m)$. Like the Task formulation, this formulation is not sensitive to the length T of the scheduling horizon.

Similarly to the Task formulation, we have the following constraints for anti-symmetry and transitivity of the precedences:

$$\neg y_{i,j} \vee \neg y_{j,i} \qquad \forall A_i, A_j \in A, i \neq j \tag{7.40}$$

$$(y_{i,j} \wedge y_{j,k}) \rightarrow y_{i,k} \qquad \forall A_i, A_j, A_k \in A, i \neq j, j \neq k, i \neq k \tag{7.41}$$

7.5.4 Event Formulation

In this formulation, some variables are indexed by events denoting, e.g., the time when such event occurs. In the RCPSP, events usually correspond to the start or to the end of an activity. In an event-based encoding there must be at least one event for each activity, but one could also consider some additional events. It can also be the case that two or more events take place at the same time.

We consider both the start and the end of an activity $A_i \in A$ as an event. We can restrict the number of events to the number n of activities plus one, because we can always enforce an activity to begin at end of another activity. Moreover, since start (end) times of different activities can coincide, the number of events can be considered to be strictly less than $n + 1$ (for example, two starts of activities taking place at the same time can be considered to be the same event, instead of two different events occurring at the same time). We take $\varepsilon = \{0, 1, \ldots, n\}$ as the index set of the events (since there will be at most $n + 1$ distinct start/end events).

In this encoding, like in the previous two, the number of variables and constraints is independent of the length T of the scheduling horizon. Several event-based encodings have been proposed in [ZHR08] and [KALM11]. Ours follows the spirit of this last one.

The integer variables t_e denote the time unit at which event e occurs. The Boolean variables $x_{e,i}$ denote if the start of activity $A_i \in A$ corresponds to event e. The Boolean variables $y_{e,i}$ denote if the end of activity $A_i \in A$ corresponds to event e. The integer variables $r_{e,k}$ denote the amount of resource k used at time unit t_e by all activities running at that moment.

$$t_0 = 0 \tag{7.42}$$

$$t_{e+1} > t_e \qquad \forall e \neq n \in \varepsilon \tag{7.43}$$

$$\bigvee_{e \in \varepsilon} x_{e,i} \qquad \forall A_i \in A \tag{7.44}$$

$$\bigwedge_{e \in \varepsilon} \bigwedge_{f > e \in \varepsilon} \neg x_{e,i} \vee \neg x_{f,i} \qquad \forall A_i \in A \tag{7.45}$$

$$\bigvee_{e \in \varepsilon} y_{e,i} \qquad \forall A_i \in A \tag{7.46}$$

$$\bigwedge_{e \in \varepsilon} \bigwedge_{f > e \in \varepsilon} \neg y_{e,i} \vee \neg y_{f,i} \qquad \forall A_i \in A \tag{7.47}$$

$$x_{e,i} \leftrightarrow t_e = S_i \qquad \forall A_i \in A, \forall e \neq n \in \varepsilon \tag{7.48}$$

$$y_{e,i} \leftrightarrow t_e = S_i + p_i \qquad \forall A_i \in A, \forall e \neq 0 \in \varepsilon \tag{7.49}$$

$$r_{0,k} = \sum_{A_i \in A} b_{i,k} * ite(x_{0,i}, 1, 0) \qquad \forall B_k \in B \tag{7.50}$$

$$r_{e,k} = r_{e-1,k} + \sum_{A_i \in A} b_{i,k} * ite(x_{e,i}, 1, 0) - \sum_{A_i \in A} b_{i,k} * ite(y_{e,i}, 1, 0) \qquad \forall B_k \in B, \forall e \in \varepsilon, 0 < e < n \tag{7.51}$$

$$r_{e,k} \leq B_k \qquad \forall B_k \in B, \forall e \in \varepsilon, 0 \leq e < n \tag{7.52}$$

Constraints (7.42) and (7.43) order chronologically the events. An important difference with previously known event-based encodings is the strict inequality in (7.43). Since different activities can start or end at the same time, these constraints are implicitly stating that different starts or ends of activities are considered to be the same event. Hence, if the number of distinct start/end times turns out to be less than $n + 1$, in our encoding there will be "extra" events that can take place at any time (i.e., the variable t_e is free to get any value) and do not correspond neither to the start nor the end of any activity in A.

Constraints (7.44), (7.45), (7.46) and (7.47) are exactly-one constraints for the $x_{e,i}$ and $y_{e,i}$ variables. Constraints (7.48) and (7.49) relate the $x_{e,i}$ and $y_{e,i}$ variables with the time of the corresponding events. Finally, constraints (7.50), (7.51) and (7.52) encode the resource constraints (7.3) at each event using the $x_{e,i}$ and $y_{e,i}$ variables.

This encoding involves $2(n+2)n$ Boolean variables, $(n+1)m$ integer variables, and $1+n+2(n+n(n-1))+2n^2+2nm$ constraints, i.e., the number of variables and constraints to add –to the ones already in (7.4) to (7.7)– is $\mathcal{O}(n^2+nm)$.

The performance of this encoding has been really disappointing (see Section 7.6.2), so we have developed a completely new encoding also based on events but exploiting in a better way the characteristics of SMT solvers. This new encoding is described in the next subsection.

7.5.5 New Event Formulation

In our new event-based encoding, we only take into account the start time of the activities, and we restrict the number of events to the number n of activities, being $\varepsilon = \{0, 1, \ldots, n-1\}$ the index set of the events. We recall from Section 7.5.2 that checking only that there is no overload at the beginning (end) of each activity is sufficient to ensure that there is no overload at every time unit. Therefore, in this formulation an event can be seen as a *resource check*, i.e., checking that the amount of resources required by all activities running at a certain time does not exceed resource availability.

For this formulation we introduce a new preprocessing, to reduce the window of events in which an activity can start, which initially is $[0, n-1]$. This preprocessing gives us an event window of the form $[ESE_i, LSE_i]$ for each activity $A_i \in A$, being ESE_i the earliest possible event for the start of A_i and LSE_i the latest possible one. To compute the event windows we use the extended event set EE^*. This set is similar to E^* (see Section 7.3) but for events. Since the precedence between events is a transitive relation, we can compute the minimum number of events between each pair of activities. For this calculation we use the Floyd-Warshall algorithm on the graph defined by the precedence relation E labeling each arc (A_i, A_j) with 1. This extended event set is named EE^* and contains, for each pair of activities A_i and A_j such that A_i precedes A_j, a tuple of the form $(A_i, A_j, l_{i,j})$ where $l_{i,j}$ is the length of the longest path of events from A_i to A_j (note that here we are also considering the dummy activities A_0 and A_{n+1}). Once computed the extended event set we can calculate the event window $[ESE_i, LSE_i]$ for the activities $A_i, 1 \leq i \leq n$, taking:

$$\begin{array}{rcll} ESE_i & = & l_{0,i} - 1 & (A_0, A_i, l_{0,i}) \in EE^* \\ LSE_i & = & n - l_{i,n+1} & (A_i, A_{n+1}, l_{i,n+1}) \in EE^* \end{array}$$

Note that after EE^* has been computed, both $(A_0, A_i, l_{0,i})$ and $(A_i, A_{n+1}, l_{i,n+1})$ are in EE^* for all A_i. Following the example of Figure 7.1, activity A_4 has the event window $[1, 4]$.

Now we present our encoding, based on the previously computed event windows. As before, the integer variables t_e denote the time unit at which event e occurs. The

Boolean variables $z^1_{e,i}$ denote if activity A_i starts at or before event e. The Boolean variables $z^2_{e,i}$ denote if activity A_i ends before event e. The integer variables $z_{e,i}$ are 1 if activity A_i is active at event e, and 0 otherwise.

$$t_0 = 0 \tag{7.53}$$

$$t_{e+1} > t_e \qquad \forall e \neq n-1 \in \varepsilon \tag{7.54}$$

$$z^1_{LSE_i,i} \qquad \forall A_i \in A \tag{7.55}$$

$$(S_i = t_{ESE_i}) \leftrightarrow z^1_{ESE_i,i} \qquad \forall A_i \in A \tag{7.56}$$

$$(S_i = t_e) \leftrightarrow (z^1_{e,i} \wedge \neg z^1_{e-1,i}) \qquad \forall A_i \in A, \forall e \in \{ESE_i + 1, \ldots, LSE_i\} \tag{7.57}$$

$$(t_e \geq S_i + p_i) \leftrightarrow z^2_{e,i} \qquad \forall A_i \in A, \forall e \in \{ESE_i, \ldots, LSE_i\} \tag{7.58}$$

$$ite(z^1_{e,i} \wedge \neg z^2_{e,i};\ z_{e,i} = 1;\ z_{e,i} = 0) \qquad \forall A_i \in A, \forall e \in \{ESE_i, \ldots, LSE_i\} \tag{7.59}$$

$$\left(\sum_{A_i \in A, e \in \{ESE_i, \ldots, LSE_i\}} b_{i,k} * z_{e,i} \right) \leq B_k \qquad \forall B_k \in B \tag{7.60}$$

Constraints (7.53) and (7.54) order chronologically the events. As before, the strict inequality in (7.54) can force some events (in this case, *resource checks*) to take place not necessarily at the start of an existing activity. Constraints (7.55), (7.56) and (7.57) define the $z^1_{e,i}$ variables and state that each start of an activity corresponds to an event. Constraints 7.58 define the $z^2_{e,i}$ variables, while constraints (7.59) define the $z_{e,i}$ variables. Finally, constraints (7.60) encode the resource constraints (7.3) at each event using the $z_{e,i}$ variables.

This encoding involves $2\sum_{i \in 1..n}(LSE_i - ESE_i + 1)$ Boolean variables, $n + \sum_{i \in 1..n}(LSE_i - ESE_i + 1)$ integer variables, and $(n+1) + n + 3\sum_{i \in 1..n}(LSE_i - ESE_i + 1) + m$ constraints, i.e., the number of variables and constraints to add –to the ones already in (7.4) to (7.7)– is $\mathcal{O}(n^2)$ and $\mathcal{O}(n^2 + m)$, respectively.

Moreover, in order to improve propagation, we add the following redundant constraints:

$$(t_e \geq S_i) \leftrightarrow z^1_{e,i} \qquad \forall A_i \in A, \forall e \in \{ESE_i, \ldots, LSE_i\} \tag{7.61}$$

$$(t_e = S_i) \rightarrow (z_{e,i} = 1) \qquad \forall A_i \in A, \forall e \in \{ESE_i, \ldots, LSE_i\} \tag{7.62}$$

$$\neg z^2_{ESE_i,i} \qquad \forall A_i \in A \tag{7.63}$$

$$z^2_{e,i} \rightarrow z^2_{e+1,i} \qquad \forall A_i \in A, \forall e \in \{ESE_i, \ldots, LSE_i - 1\} \tag{7.64}$$

$$z^1_{e,i} \rightarrow z^1_{e+1,i} \qquad \forall A_i \in A, \forall e \in \{ESE_i, \ldots, LSE_i - 1\} \tag{7.65}$$

$$z^1_{e,i} \rightarrow z^2_{e+p_i-1,i} \qquad \forall A_i \in A, \forall e \in \{ESE_i, \ldots, LSE_i - p_i + 1\} \tag{7.66}$$

Redundant constraints, in particular the ones of (7.61), have significantly improved the performance of our solver in this formulation.

7.6 Experiments

In this section we discuss the results of our experiments.

- First, we check the performance of the initial and the new event-based formulation.
- Second, we check the effect of adding the different preprocesses proposed in Section 7.3 in all the presented encodings. We make these experiments considering all the optimization methods of Section 7.4.
- Third, we compare the efficiency of our best solving configuration (i.e., the best combination of encoding, preprocesses and optimization methods) with the best (exact) system currently known, Lazy_fd [SFSW09]. Additionally we check the robustness of both systems when incrementing the timeout to 3600 seconds.
- Fourth, we check the effect of adding several improvements in our system `rcpsp2smt` and compare again its performance with the one of Lazy_fd. We also check the robustness of these improvements when incrementing the timeout to 3600 seconds.

We have run the experiments on an Intel® Core™ i5 CPU at 2.66 GHz, with 6GB of RAM, running 64-bit openSUSE 11.2 (kernel 2.6.31). We have used Yices 1.0.34 as the core SMT solver. The timeout was set to 500 seconds for all experiments, except for large timeout tests in which we mention the timeout explicitly. The instances we have used are from the PSPLib and from [KALM11] and [CN03]. The solving times that we report for our tool include the preprocessing and encoding time.

7.6.1 Initial and New Event-Based Formulation

In Table 7.1 we present the results of the experiments to compare the first event-based encoding and the new one. In this experiments we have used the j60 package from the PSPLib without any kind of preprocessing. We can see that the performance of the new encoding is much better than the one of the initial encoding. Using the initial encoding only 81 instances were solved within the timeout, while using the new encoding the number of solved instances raised up to 348 of 480.

In the remaining comparisons of the section, by event formulation we refer to the new event-based formulation.

	Initial Formulation	New Formulation
#Solved	81	348
Mean	275.55	30.95
Median	259.18	19.76

Table 7.1: Comparison of the initial event-based formulation and the new one, showing the number of solved instances, and the mean and the median of the solving times (cutoff 500 seconds).

7.6.2 Preprocessing and Optimization

Table 7.2 presents the results of the experiments using the four formulations presented in the encodings section (Section 7.5): time, task, flow and event. In this experiments we have used the j60 package from the PSPLib. We consider that this package is quite representative, since it contains instances of different levels of difficulty. For each formulation, we have considered all the possible combinations of the three different preprocessing techniques (Section 7.3):

- using the heuristic method or the trivial one for computing the upper bound,
- using or not the extended precedence set, and
- using or not the incompatibility matrix.

The use of the heuristic computation of the upper bound is better than using the trivial upper bound. In all encodings, a better upper bound reduces the domains of the start variables (S_i), and in the case of the time encoding it also considerably reduces the number of t_i variables (see Subsection 7.5.1). The benefit of using the other techniques is not always clear, in fact, in some cases it turns to be counterproductive. Nevertheless, it seems that the use of the three preprocessing techniques together, is the best configuration for all encodings.

We can also see that the best encoding is always the time one, followed by the task and event ones, and far away by the flow encoding.

In Table 7.3 we present the results of the experiments made to check all the optimization methods that we have implemented (see Section 7.4). We have considered the same set of instances as before. In these experiments we have only considered the best encoding and configuration of preprocesses from the previous experiments: the Time formulation with extended precedences, incompatibility matrix and heuristic upper bound.

As we can see, there is no much difference between the different tested types of optimization, since all of them solve the same number of instances (430). However, it

	Time							
	Heuristic U.B.				Trivial U.B.			
	Ext. Prec.		Not Ext. Prec.		Ext. Prec.		Not Ext. Prec.	
	Inc.	N. Inc.	Inc.	N. Inc.	Inc.	N. Inc.	Inc.	N. Inc.
#Solved	430	430	429	429	430	429	429	430
Mean	4.53	5.52	6.30	3.60	11.89	15.03	11.08	10.36
Median	0.20	0.21	0.69	0.19	5.03	5.75	5.65	3.95

	Task							
	Heuristic U.B.				Trivial U.B.			
	Ext. Prec.		Not Ext. Prec.		Ext. Prec.		Not Ext. Prec.	
	Inc.	N. Inc.	Inc.	N. Inc.	Inc.	N. Inc.	Inc.	N. Inc.
#Solved	398	396	397	396	398	396	398	397
Mean	13.91	13.16	10.86	11.44	27.53	28.64	20.99	34.66
Median	4.58	5.5	3.45	4.66	18.1	23.04	12.01	26.54

	Flow							
	Heuristic U.B.				Trivial U.B.			
	Ext. Prec.		Not Ext. Prec.		Ext. Prec.		Not Ext. Prec.	
	Inc.	N. Inc.	Inc.	N. Inc.	Inc.	N. Inc.	Inc.	N. Inc.
#Solved	272	261	239	263	152	138	118	148
Mean	38.58	32.66	21.88	29.87	83.26	78.02	25.55	86.13
Median	0.01	0.01	0.01	0.01	14.85	23.42	17.45	19.21

	Event							
	Heuristic U.B.				Trivial U.B.			
	Ext. Prec.		Not Ext. Prec.		Ext. Prec.		Not Ext. Prec.	
	Inc.	N. Inc	Inc	N. Inc.	Inc.	N. Inc.	Inc.	N. Inc
#Solved	349	349	348	348	349	348	348	348
Mean	21.55	17.87	15.90	14.74	37.02	48.75	30.09	30.95
Median	7.22	5.79	4.87	4.57	25.39	35.24	19.81	19.76

Table 7.2: Comparison of the different proposed encoding with the different preprocessing techniques (cutoff 500 seconds).

seems that the search procedures (dico, hybrid, linear) are slightly better than the ones of Weighted SMT (yices, wpm1, wbo).

	dico	hybrid	linear	yices	wpm1	wbo
#Solved	430	430	430	430	430	430
Mean	5.01	4.53	4.34	6.68	7.25	6.26
Median	0.18	0.20	0.19	0.18	0.21	0.27

Table 7.3: Comparison of the different optimization methods (cutoff 500 seconds).

7.6.3 Comparison with Others Solvers

We focus this experiment on 7 families of RCPSP instances: j30, j60, j90 and j120 (from the PSPLib), KS15_d and Pack_d from [KALM11], and Pack from [CN03].

	j30 (480 inst.)		j60 (480 inst.)		j90 (480 inst.)		j120 (600 inst.)	
	Time	Lazy_fd	Time	Lazy_fd	Time	Lazy_fd	Time	Lazy_fd
#Solved	480	480	430	430	395	396	274	277
Mean	0.65	0.21	4.53	3.92	5.14	4.17	12.68	9.30
Median	0.08	0.01	0.20	0.03	0.39	0.06	1.98	0.20

	KS15_d (479 inst.)		Pack_d (55 inst.)		Pack (55 inst.)		Total (2629 inst.)	
	Task	Lazy_fd	Task	Lazy_fd	Time	Lazy_fd	rcpsp2smt	Lazy_fd
#Solved	479	479	28	37	39	16	2125	2115
Mean	0.03	0.01	61.04	36.15	23.44	107.34	4.89	4.29
Median	0.02	0.01	0.72	3.34	1.07	66.88	0.10	0.03

Table 7.4: Comparison between our best approach and Lazy_fd (cutoff 500 seconds).

In Table 7.4 we can see the summary of the comparison between our best approach and one of the best current approaches: the one of the Lazy_fd solver from [SFSW09].

Note that we always use the Time encoding, except in KS15_d and Pack_d sets, where the Task encoding obtained the best results. This may be due to the structure of this set with long duration activities, which makes the scheduling horizon to be very large, so that the number of variables in the Time formulation is very high in contrast to the encoding Task which always uses $\mathcal{O}(n^2)$ variables. The same complexity applies to the number of constraints.

It can be seen that the two approaches have similar performance: both solve the same number of instances in the j30 and j60 sets, and Lazy_fd solves one more instance in the j90 set and three more instances in the j120 set. The KSD_15 set turns out to be very easy for the two approaches. The Pack and Pack_d sets contain the same instances,

differing only on the durations of activities, that in Pack_d are much larger. We see that Lazy_fd performs better with longer durations. However, `rcpsp2smt` performs better in the case of the whole pack with very few precedences and small durations. To sum up, with our approach we solve ten more instances than Lazy_fd, mainly due to the Pack set. In general, Lazy_fd always gets a smaller mean and median time in all sets. It is worth noting that we have chosen our best encoding (either Time or Task formulation) for each set, instead of a fixed one. This can be considered unfair, but we think that it illustrates better the strengths and weaknesses of each approach.

	j30		j60		j90		j120	
	(480 inst.)		(480 inst.)		(480 inst.)		(600 inst.)	
	Time	Lazy_fd	Time	Lazy_fd	Time	Lazy_fd	Time	Lazy_fd
#Solved	480	480	437	432	404	402	285	283
Mean	0.65	0.21	28.63	9.06	41.72	30.97	48.21	21.38
Median	0.08	0.01	0.21	0.03	0.39	0.06	1.86	0.20

	KS15_d		Pack_d		Pack		Total	
	(479 inst.)		(55 inst.)		(55 inst.)		(2629 inst.)	
	Task	Lazy_fd	Time	Lazy_fd	Time	Lazy_fd	rcpsp2smt	Lazy_fd
#Solved	479	479	38	38	41	18	2164	2132
Mean	0.03	0.01	394.86	89.99	60.52	332.49	28.15	14.97
Median	0.02	0.01	50.55	4.05	1.81	84.80	0.11	0.03

Table 7.5: Comparison between our approach and Lazy_fd (cutoff 3600 seconds).

In Table 7.5 we can see the comparison with a bigger time-out (one hour). In this case the Time encoding is better than Task encoding for the Pack_d set. In the j90 and j120 sets, where Lazy_fd solved 1 and 3 more instances, respectively, with a cutoff of 500 seconds, now `rcpsp2smt` turns out to be the best approach, with 2 more solved instances than Lazy_fd in each set. The Pack_d set is now balanced, with 38 solved instances by the two approaches and, finally, `rcpsp2smt` keeps the advantage over Lazy_fd in the Pack set with 23 more solved instances. However, Lazy_fd continues having better mean and median times than `rcpsp2smt`.

7.6.4 System Improvements

Once seen that preprocessing is useful and that the best way to achieve optimization is using iterative search methods, we have tried to improve the performance of our system by introducing some additional improvements in this two issues. First, we have added a new preprocessing step and, second, we have modified the search algorithm in such a way

that when it founds a solution, it tries to improve the time windows of activities before the next step of the search algorithm is executed.

The preprocessing step is the following:

- *No holes*. We have observed that many solutions contain holes. Holes occur when the start of an activity does not immediately follow the end of another activity. Obtaining solutions without these holes can result in better makespans. In order to obtain such solutions we have added the following constraints:

$$\bigvee_{\substack{A_j \in A \cup \{A_0\} \setminus \{A_i\} \\ (A_i, A_j, l_{i,j}) \notin E^* \\ (A_j, A_i, l_{j,i}) \notin E^* \vee (A_j, A_i) \in E}} S_i = S_j + p_j \quad \forall A_i \in A \cup \{A_{n+1}\}$$

  The results obtained with these new restrictions are not much better than the ones obtained without them. The only constraints that seem to improve the performance of the system are the ones related to A_0. Therefore, we finally decided to add only the following constraint:

$$\bigvee_{(A_0, A_i) \in E} S_i = 0 \quad \forall A_i \in A \cup \{A_{n+1}\}$$

The improvements on the time windows of activities come from the following two computations:

- *Active schedule search*. We have observed that most of the solutions found in the intermediate steps of the search algorithm are not active. A schedule is active [SKD95, ADN07] if it admits no feasible *activity global left shift*. A global left shift operator $GLS(S, A_i, \Delta)$ transforms schedule S into an schedule S', where the start of activity A_i has been left shifted Δ units of time, i.e., $S'_i = S_i - \Delta$ with $\Delta > 0$. Sometimes, by turning a solution into active, we can improve the makespan.

  To obtain active solutions from intermediate solutions, we iteratively apply the global left shift operator. We begin with the activities with the smallest start time. If a lower makespan is found, then we use this as the new upper bound.

  In the Figure 7.2 we can see some non-active schedules and their equivalent active schedules (the RCPSP instance is the same of Figure 7.1). By sequentially applying the GLS operator to activities 5, 3, 6 and 7 with Δ equal to 5, 3, 3 and 3, respectively, the makespan decreases in 3 units.

- *Time windows adequacy*. When using search algorithms for optimization, we are able to find a new upper bound at each execution of the decision procedure. In

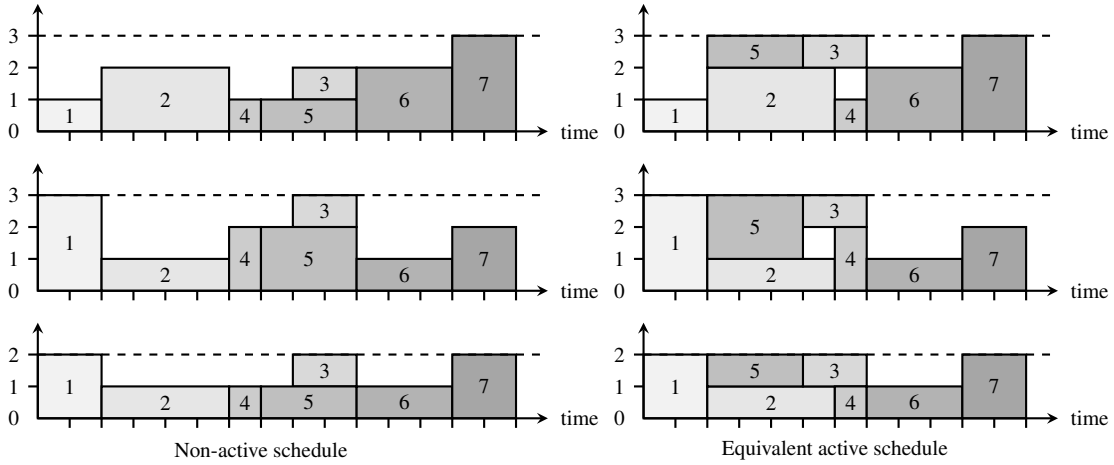


Figure 7.2: An example of active schedule.

addition to narrowing the search thanks to better upper bounds, we can redefine all time windows. Thus, we also reduce the domains of the start variables S_i. For activities $A_i \in A$, we have

$$newLS_i = newUB - l_{i,n+1}$$

where $(A_i, A_{n+1}, l_{i,n+1}) \in E^*$. For activity A_{n+1},

$$newLS_{n+1} = newUB$$

The constraints added in each step are:

$$S_i \leq newLS_i \quad \forall A_i \in A \cup \{A_{n+1}\}$$

Note that in the Time formulation, when we reduce the time windows of activities, many variables can be set to 0. To this end, we add the following constraints:

$$x_{i,t} = 0 \quad \forall A_i \in A \cup \{A_{n+1}\}, \forall t \in \{LS_i + p_i, \ldots, prevLS_i + p_i - 1\}$$

where $prevLS_i$ is the latest start time of A_i, computed in the previous step of the search algorithm.

7.6.5 Closed Instances

With the changes outlined in subsection 7.6.4 we have been able to close the PSPLib instances enumerated in Table 7.6.

	Instance	Makespan	Encoding	Solver	Time
j60	9_3	100	Time	hybrid	1365.20
	9_8	96	Time	linear	2019.23
	9_9	99	Time	hybrid	668.60
	25_4	108	Time	hybrid	2501.72
	25_5	98	Time	hybrid	460.19
	25_10	108	Time	hybrid	1033.83
	30_2	70	Time	hybrid	832.13
j90	5_4	102	Time	hybrid	2574.40
	5_6	86	Time	hybrid	2340.54
	5_9	115	Time	hybrid	2701.99
	21_1	110	Time	hybrid	1948.87
	26_5	85	Time	hybrid	642.93
	37_2	115	Time	hybrid	2329.42
	46_4	93	Time	hybrid	1933.28
j120	1_1	105	Time	hybrid	1803.32
	8_3	95	Time	hybrid	235.95
	8_6	85	Time	hybrid	1983.62(*)
	48_5	110	Time	hybrid	684.62
	49_2	109	Time	hybrid	2339.06(*)

Table 7.6: New closed instances (cutoff 3600 seconds). (*) Solutions obtained without the improvements of Section 7.6.4.

7.7 Summary

This chapter contains one of the main objectives of this thesis. It shows that SMT is a competitive approach for solving CSP and COP in front of ad hoc algorithms. We have tested this very good performance on the well-known resource-constrained project scheduling problem. Using an algorithm with preprocessing, an iterative process of minimization, etc., and using an SMT solver as the core solving engine, we have obtained performances that are similar, or even better, than those of best state-of-the-art algorithms.

We have described the four different encodings implemented, with a a number of variants for each of them. After many tests we have reached the conclusion that the best coding in general is the classical Time formulation, closely followed by the Task formulation. In fact, the Task formulation is the best in some types of instances, concretely, the ones with a larger scheduling horizon. We have also provided a whole new event-based formulation.

Chapter 8

Other Scheduling Problems

As we have seen in Chapter 7, we have obtained very good results when using SMT to solve the RCPSP. There exist many variations of the RCPSP. In this chapter we study the performance of SMT when dealing with several of those variants. Namely, we work on the Resource-Constrained Project Scheduling Problem with minimum and maximum time lags (RCPSP/max), and the Multi-Mode Resource-Constrained Project Scheduling Problem (MRCPSP). This last variant consists in the selection of a single activity mode, from a set of available modes, in order to construct a precedence, and a resource feasible project schedule with a minimal makespan. Contrarily to RCPSP, MRCPSP considers renewable and non-renewable resources.

This chapter corresponds to the fourth objective of the thesis (to provide an SMT based system being competitive with state-of-the-art methods for scheduling problems) and provides the eight contribution.

8.1 RCPSP/max

The Resource Constrained Project Scheduling Problem with minimum and maximum time lags (RCPSP/max)[1] is a generalization of the RCPSP where the precedence graph $G(V, E)$ becomes $G(V, E, g)$, being g an edge labeling function, valuating each edge $(A_i, A_j) \in E$ with an integer time lag $g_{i,j}$. Non-negative lags $g_{i,j} \geq 0$ correspond to a minimum delay in the start of activity A_j with respect to the start of activity A_i. The case $g_{i,j} < 0$ corresponds to a maximum delay of $-g_{i,j}$ units in the start of activity A_i with respect to the start of activity A_j (see Figure 8.1). Note that the standard RCPSP can be seen as the particular case of RCPSP/max where only minimum time lags are considered, taking $g_{i,j} = p_i \; \forall (A_i, A_j) \in E$.

[1] This problem is denoted as $PS|temp|C_{max}$ in [BDM+99] and $m, 1|gpr|C_{max}$ in [HL05].

Regardless of minimizing the makespan, deciding if there exits a resource-feasible schedule that respects the minimum and maximum lags is NP-complete [BMR88], and the optimization problem is NP-hard in the general case.

The treatment that we propose for this problem is very similar to the one proposed for RCPSP. We only consider the Time and Task encodings, since these are the ones which have shown best performance in the standard RCPSP. For RCPSP/max these encodings are almost the same as the ones proposed in Section 7.5. We simply need to replace the constraints (7.2) by the following constraints, where we consider time lags instead of durations:

$$S_j - S_i \geq g_{i,j} \qquad \forall (A_i, A_j) \in E \tag{8.1}$$

The preprocessing is almost the same but with some changes in the computation of the upper and lower bounds for the makespan, as we explained in the next subsection. The solving process is identical.

8.1.1 Preprocessing

A RCPSP/max instance is unsatisfiable if and only if there is a cycle of positive length in the precedence graph. This is checked by calculating the extended precedence set E^* (which takes into account only the positive edges in the precedence graph) and checking two conditions:

1. There is no cycle: $\nexists (A_i, A_i, l_{i,i}) \in E^*$ s.t. $l_{i,i} > 0$.

2. There is no inconsistency between the maximum and the minimum time lags between activities: for every negative edge $g_{ji} < 0$ and $(A_i, A_j, l_{i,j}) \in E^*$, it must be satisfied that $|g_{ji}| >= l_{i,j}$.

Since finding a resource-feasible schedule in RCPSP/max is NP-complete, we do not use a heuristic to calculate the upper bound of the makespan. The upper bound that we consider is the trivial upper bound:

$$UB = \sum_{A_i \in A} \max(p_i, \max_{(A_i,A_j) \in E}(|g_{i,j}|))$$

The lower bound, the time windows and the incompatibilities between activities are calculated as in the RCPSP.

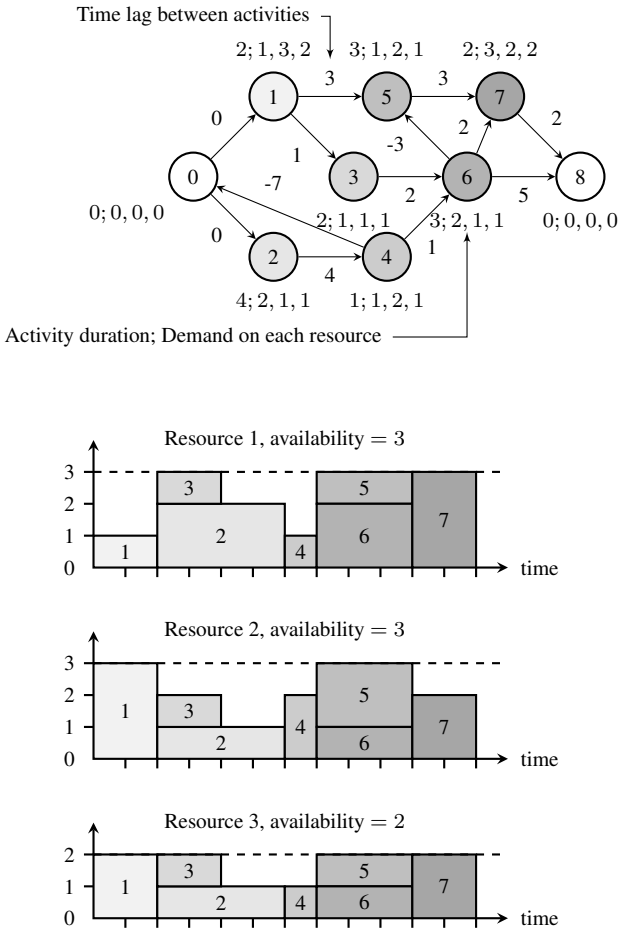


Figure 8.1: An example of RCPSP/max and one of its possible solutions.

8.1.2 Experiments

We have performed experiments on the RCPSP/max instances from the PSPLib, testsets j10, j20, j30. The experiments have been run on an Intel® Core™ i5 CPU at 2.66 GHz, with 6GB of RAM, running 64-bit openSUSE 11.2 (kernel 2.6.31). We have used Yices 1.0.34 as the core SMT solver. The timeout was set to 3600 seconds.

Table 8.1 shows a comparison between our approach with the Time formulation and the currently best known approach, the one Lazy_fd. We can observe that our system is really competitive in all sets of instances. For RCPSP/max we have not considered the system improvements presented in Subsection 7.6.4.

	j10		j20		j30	
	(270 inst.)		(270 inst.)		(270 inst.)	
	Time	Lazy_fd	Time	Lazy_fd	Time	Lazy_fd
#Solved	270	270	270	270	265	265
Mean	0.12	0.0	1.92	0.16	32.84	23.91
Median	0.10	0.0	0.78	0.01	2.75	0.03

Table 8.1: Comparison between our system rcpsp2smt and Lazy_fd, showing the number of solved instances, and the mean and median times (cutoff 3600 seconds).

8.2 Multimode RCPSP

The Multi-Mode Resource-Constrained Project Scheduling Problem (MRCPSP)[2] is an extension of the RCPSP. In this extension every activity has a number, greater or equal to 1, of execution modes. An activity mode is described with a pair formed by the duration of the activity and a vector with the resource demands of the activity for this mode. In the MRCPSP one distinguishes between renewable resources and non-renewable resources: renewable resources are replenished at each time unit, while for the non-renewable ones, resource usage is accumulated across the entire project. For example, a renewable resource could be man-hours per day, or the capacity of a machine, while a non-renewable resource could be a budget, or some kind of stock.

Thus, the objective of the MRCPSP is to find a mode and a start time for each activity such that the makespan is minimized and the schedule is feasible with respect to the precedence and renewable and non-renewable resource constraints, minimizing the makespan. The MRCPSP is NP-hard. Several exact and heuristic approaches to solve the

[2]This problem is denoted as $MPS|prec|C_{max}$ in [BDM+99] and $m, 1T|cpm, disc.mu|C_{max}$ in [HL05].

MRCPSP have been proposed in the last years. The most common exact approaches for solving this problem are based on branch-and-bound [SD98, ZBY06] and linear programming. There are some algorithms that use SAT [CV11], but none using SMT.

More formally, the MRCPSP is defined by a tuple (V, M, p, E, R, B, b) where[3]:

- $V = \{A_0, A_1, \ldots, A_n, A_{n+1}\}$ is a set of activities. Activity A_0 represents by convention the start of the schedule and activity A_{n+1} represents the end of the schedule. The set of non-dummy activities is defined by $A = \{A_1, \ldots, A_n\}$.
- $M \in \mathbb{N}^{n+2}$ is a vector of naturals, being M_i the number of modes that activity i can execute, with $M_0 = M_{n+1} = 1$ and $M_i \geq 1, \forall A_i \in A$.
- p is a vector of vectors of durations. $p_{i,o}$ denotes the duration of activity i using mode o, with $1 \leq o \leq M_i$. For the dummy activities, $p_{0,1} = p_{n+1,1} = 0$, and $p_{i,o} > 0 \; \forall A_i \in A, 1 \leq o \leq M_i$.
- E is a set of pairs representing precedence relations. Thus, $(A_i, A_j) \in E$ means that the execution of activity A_i must precede that of activity A_j, i.e., activity A_j must start after activity A_i has finished. We assume that we are given a precedence activity-on-node graph $G(V, E)$ that contains no cycles; otherwise the precedence relation is inconsistent. Since the precedence is a transitive binary relation, the existence of a path in G from the node i to node j means that activity i must precede activity j. We assume that E is such that A_0 is a predecessor of all other activities and A_{n+1} is a successor of all other activities.
- $R = \{R_1, \ldots, R_{v-1}, R_v, R_{v+1}, \ldots, R_q\}$ is a set of q resources. The first v resources are renewable, and the last $q - v$ resources are non-renewable.
- $B \in \mathbb{N}^q$ is a vector of resource availabilities. B_k denotes the available amount of each resource R_k. The first v resource availabilities are the available amounts of the renewable resources, while the last $q - v$ ones are the available amounts of the non-renewable resources.
- b is a matrix of resource demands of the activities per mode. $b_{i,k,o}$ represents the amount of resource R_k used during the execution of A_i in mode o. Note that $b_{0,k,1} = 0$ and $b_{n+1,k,1} = 0 \; \forall k \in \{1, \ldots, q\}$.

A schedule is a vector $S = (S_0, S_1, \ldots, S_n, S_{n+1})$ where S_i denotes the start time of each activity $A_i \in V$. We assume that $S_0 = 0$. A schedule of modes is a vector $SM = (SM_0, SM_1, \ldots, SM_n, SM_{n+1})$ where SM_i, satisfying $1 \leq SM_i \leq M_i$, denotes the mode of each activity $A_i \in V$. A solution of the MRCPSP problem is a schedule

[3]Although V and E are the same that in standard RCPSP, we reproduce their definitions.

of modes SM and a schedule S of minimal makespan S_{n+1}, subject to the following precedence and resource constraints:

$$\text{minimize } S_{n+1} \tag{8.2}$$

subject to:

$$SM_i = o \rightarrow S_j - S_i \geq p_{i,o} \quad \forall (A_i, A_j) \in E, \forall o \in \{1, \ldots, M_i\} \tag{8.3}$$

$$\sum_{A_i \in A,\, o \in \{1,\ldots,M_i\}} ite(SM_i = o; b_{i,k,o}; 0) \leq B_k \quad \forall R_k \in \{R_{v+1}, \ldots, R_q\} \tag{8.4}$$

$$\sum_{A_i \in A,\, o \in \{1,\ldots,M_i\}} ite(SM_i = o \wedge S_i \leq t < S_i + p_{i,o}; b_{i,k,o}; 0) \leq B_k \quad \forall R_k \in \{R_1, \ldots, R_v\}, \forall t \in H \tag{8.5}$$

where the set $H = \{0, \ldots, T\}$ is the scheduling horizon, and T (the length of the scheduling horizon) is an upper bound for the makespan.

A schedule S is feasible if it satisfies the generalized precedence constraints (8.3), the non-renewable resource constraints (8.4) and the renewable resource constraints (8.5). An example is shown in Figure 8.2.

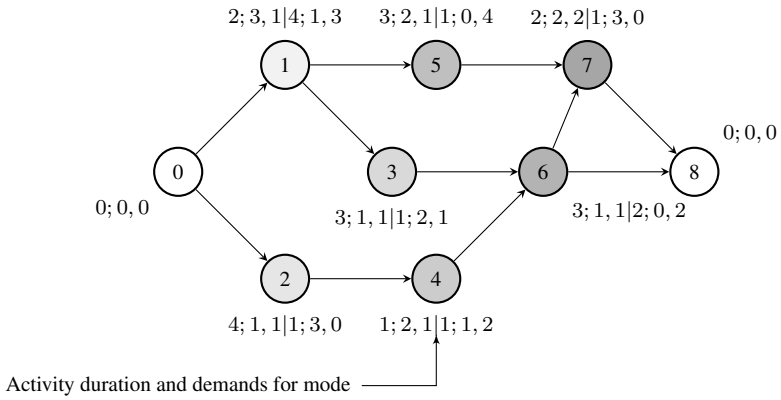
The treatment that we propose for this problem is similar to the one that we have presented for the RCPSP. However, there are many changes in the preprocessing phase and in the two encodings considered (Time and Task). The solving process is the same as in the RCPSP.

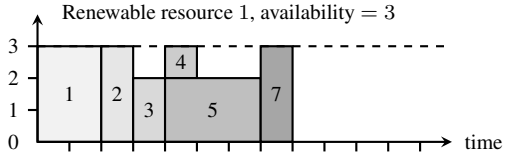
8.2.1 Preprocessing

We apply all the preprocessing methods described for the RCPSP, with appropriate modifications, except for the calculation of incompatibilities. We have also implemented an additional preprocessing method to reduce the demand of non-renewable resources.

Extended Precedence Set

In the MRCPSP, if there is a precedence between two activities, we can only ensure that the distance between the start of A_i and the start of A_j will be greater than or equal to the minimum duration of the different modes of A_i. Since a precedence is a transitive relation we can compute the minimum precedence between each pair of activities.





Non-renewable resource 2, availability $= 8$

$$1 + 0 + 1 + 2 + 1 + 2 + 0 \leq 8$$

$$SM = (1, 1, 2, 2, 2, 1, 2, 2, 1)$$

Figure 8.2: An example of MRCPSP with one renewable resource and one non-renewable resource.

Similarly to the RCPSP, to calculate the extended precedence set, we use the Floyd-Warshall algorithm on the graph defined by the precedence relation E, where each arc (A_i, A_j) is labeled with the duration $\min_{o \in \{1,\ldots,M_i\}}(p_{i,o})$. This extended precedence set is named E^* and contains, for each pair of activities A_i and A_j such that A_i precedes A_j, a tuple of the form $(A_i, A_j, l_{i,j})$ where $l_{i,j}$ is the length of the longest path from A_i to A_j. Notice also that, if $(A_i, A_i, l_{i,i}) \in E^*$ for some A_i and $l_{i,i} > 0$, then there is a cycle in the precedence relation and therefore the problem is inconsistent and has no solution.

Lower Bound

We have only implemented the *critical path bound* on the graph defined by the precedence relation E where each arc (A_i, A_j) is labeled with the duration $\min_{o \in \{1,\ldots,M_i\}}(p_{i,o})$. Notice that we can easily know the length of this path if we have already computed the precedence set, since we only need to obtain $l_{0,n+1}$ from $(A_0, A_{n+1}, l_{0,n+1}) \in E^*$.

For instance, in Figure 8.2 the critical path is $[A_0, A_1, A_3, A_6, A_7, A_8]$ with modes

$1, 1, 2, 2, 2, 1$ respectively, and its length is 6.

Upper Bound

We calculate the *trivial upper bound*, i.e., the sum of the maximum duration for all activities:

$$UB = \sum_{A_i \in A} \max_{o \in \{1,\ldots,M_i\}} (p_{i,o})$$

For instance, in the example of Figure 8.2 the upper bound is 20.

Time Windows

This preprocessing is the same as in the standard RCPSP.

Non-Renewable Resource Demand Reduction

This preprocessing reduces the demand of non-renewable resources for each activity. This allows us to save SMT literals in the non-renewable resource constraints.

For instance, in the example of Figure 8.2, the non-renewable resource R_2 has 8 units available and activity A_6 has two modes: mode 1 requires 1 unit of resource R_2, while mode 2 requires 2 units of the same resource. This problem can be transformed into an equivalent one, where the availability of resource R_2 is 7, and activity A_6 has a demand of 0 units of resource R_2 in mode 1, and of 1 unit in mode 2. Since in mode 1 the demand is of 0 units, it is not necessary to add any literal considering this mode into the non-renewable resource constraints. Roughly, following the example, what is done is to substract the minimum amount of resource R_2 that A_6 will need, from the availability of the resource and from the different demands of the activity in each mode. Clearly, this transformation is sound in the sense that it preserves the set of solutions.

In order to perform this preprocess, we construct a new vector B' of resource availabilities and a new matrix b' of resource demands of activities, where:

- B'_k and $b'_{i,k,o}$ are identical to B_k and $b_{i,k,o}$ $\forall A_i \in V, \forall o \in \{1, \ldots, M_i\}, \forall k \in \{1, \ldots, v\}$.

- For each non-renewable resource R_k and activity A_i, we find the demand value with more occurrences for the resource R_k in the different modes of this activity; we call

this value $max_{k,i}$. Then we state:

$$b'_{i,k,o} = b_{i,k,o} - max_{k,i} \qquad \forall A_i \in A, \forall o \in \{1, \ldots, M_i\}$$
$$B'_k = B_k - \sum_{A_i \in A} max_{k,i} \qquad \forall R_k \in \{R_{v+1}, \ldots, R_q\}$$

In Subsection 8.2.4 we show the good performance of this preprocessing method.

8.2.2 Encodings

In all encodings We use the set of integer variables $\{S_0, S_1, \ldots, S_n, S_{n+1}\}$ to encode the schedule. By S' we denote the set $\{S_1, \ldots, S_n\}$. We also use the set of integer variables $\{SM_0, SM_1, \ldots, SM_n, SM_{n+1}\}$ to encode the schedule of modes. The objective function is always (8.2), and we have the following constraints:

(8.6) $S_0 = 0$

(8.7) $S_i \geq ES_i \qquad \forall A_i \in \{A_1, \ldots, A_{n+1}\}$

(8.8) $S_i \leq LS_i \qquad \forall A_i \in \{A_1, \ldots, A_{n+1}\}$

(8.9) $S_j - S_i \geq l_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^*$

(8.10) $SM_0 = 1$

(8.11) $SM_{n+1} = 1$

(8.12) $SM_i \geq 1 \qquad \forall A_i \in A$

(8.13) $SM_i \leq M_i \qquad \forall A_i \in A$

where (8.7) and (8.8) are simple encodings of the time windows for each activity, (8.9) encodes the extended precedences and (8.10), (8.11),(8.12) and (8.13) encode the possible modes for each activity.

We also have the non-renewable resource constraints using the values resulting from the non-renewable resource demand reduction preprocessing:

(8.14) $$\sum_{A_i \in A, o \in \{1,\ldots,M_i\}} ite(SM_i = o; b'_{i,k,o}; 0) \leq B'_k \qquad \forall R_k \in \{R_{v+1}, \ldots, R_q\}$$

Hence, in each encoding, besides their own, there are $1 + 2(n+1) + |E^*| + 2 + 2n$ –i.e. $\mathcal{O}(n^2)$– constraints.

Time Formulation

Recall that for every time unit and renewable resource, the sum of all resource requirements for the activities in their current modes must be less than or equal to the resource availability. We use the following equations to encode these constraints:

$$ite((S_i \leq t) \wedge \neg(S_i \leq t - p_{i,o}) \wedge (SM_i = o);\ x_{i,t} = 1;\ x_{i,t} = 0) \quad (8.15)$$
$$\forall S_i \in S', \forall o \in \{1, \ldots, M_i\}, \forall t \in \{ES_i, \ldots, LS_i + \max_{o' \in \{1,\ldots,M_i\}}(p_{i,o'}) - 1\}$$

$$\sum_{A_i \in A, o \in \{1,\ldots,M_i\}} ite(t \in \{ES_i, \ldots, LS_i + p_{i,o} - 1\} \wedge (SM_i = o);\ b'_{i,r,o} * x_{i,t};\ 0) \leq B'_r \quad (8.16)$$
$$\forall R_r \in \{R_1, \ldots, R_v\}, \forall t \in H$$

We remark that (8.15) imposes that $x_{i,t} = 1$ if activity A_i is active at time t with mode o, and $x_{i,t} = 0$ otherwise. We restrict the possible time units to check by using the time windows. Constraints (8.16) encode the renewable resource constraints (8.5) using these $x_{i,t}$ variables.

This encoding involves $\sum_{A_i \in A}(LS_i + \max_{o \in \{1,\ldots,M_i\}}(p_{i,o}) - ES_i)$ –i.e. $\mathcal{O}(nT)$– new integer variables. It also involves $\sum_{A_i \in A}(LS_i + \max_{o \in \{1,\ldots,M_i\}}(p_{i,o}) - ES_i)M_i + (T+1)v$ –i.e. $\mathcal{O}(T(nM_{max} + v))$– new constraints, where $M_{max} = max_{A_i \in A}(M_i)$.

We can slightly change this previous formulation by introducing $\mathcal{O}(nT)$ Boolean variables $y_{i,t}$ encoding the $0/1$ $x_{i,t}$ variables, by replacing the constraints (8.15) with the following constraints:

$$y_{i,t} \leftrightarrow (S_i \leq t) \wedge \neg(S_i \leq t - p_{i,0}) \wedge (SM_i = o) \quad (8.17)$$
$$\forall S_i \in S', \forall o \in \{1, \ldots, M_i\}, \forall t \in \{ES_i, \ldots, LS_i + \max_{o' \in \{1,\ldots,M_i\}}(p_{i,o'}) - 1\}$$

$$ite(y_{i,t};\ x_{i,t} = 1;\ x_{i,t} = 0) \quad (8.18)$$
$$\forall S_i \in S', \forall t \in \{ES_i, \ldots, LS_i + \max_{o \in \{1,\ldots,M_i\}}(p_{i,o}) - 1\}$$

We have observed that for small problem instances this last formulation exhibits better performance than the previous one. However, for big instances (with more than 50 activities) the addition of the new variables $y_{i,t}$ usually makes the problem intractable by our system.

Task Formulation

Similarly as in Subsection 7.5.2, in this formulation we use variables indexed by activities instead of by time. The key idea is that checking only that there is no overload at the

beginning (end) of each activity is sufficient to ensure that there is no overload at every time (for the non-preemptive case). In this formulation, the number of variables and constraints is independent of the length of the scheduling horizon T.

The constraints are the following, where the Boolean variables $z^1_{i,j}$ denote if activity A_i starts not after A_j. The Boolean variables $z^2_{i,j}$ denote if activity A_j starts before A_i ends. The integer variables $z_{i,j}$ are 1 if activity A_i is active when activity A_j starts, and 0 otherwise.

$$z^1_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{8.19}$$

$$\neg z^1_{j,i} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{8.20}$$

$$z^1_{i,j} \leftrightarrow S_i \leq S_j \qquad \begin{array}{l} \forall A_i, A_j \in A, \\ (A_i, A_j, l_{i,j}) \notin E^*, (A_j, A_i, l_{j,i}) \notin E^*, \\ i \neq j \end{array} \tag{8.21}$$

$$\neg z^2_{i,j} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{8.22}$$

$$z^2_{j,i} \qquad \forall (A_i, A_j, l_{i,j}) \in E^* \tag{8.23}$$

$$SM_i = o \rightarrow (z^2_{i,j} \leftrightarrow S_j < S_i + p_{i,o}) \qquad \begin{array}{l} \forall A_i, A_j \in A, \forall o \in \{1, \ldots, M_i\}, \\ (A_i, A_j, l_{i,j}) \notin E^*, (A_j, A_i, l_{j,i}) \notin E^*, \\ i \neq j \end{array} \tag{8.24}$$

$$ite(z^1_{i,j} \wedge z^2_{i,j};\ z_{i,j} = 1;\ z_{i,j} = 0) \qquad \begin{array}{l} \forall A_i, A_j \in A, \\ i \neq j \end{array} \tag{8.25}$$

$$SM_j = o' \rightarrow \sum_{A_i \in A \setminus \{A_j\}, o \in \{1, \ldots, M_i\}} ite(SM_i = o; b'_{i,k,o} * z_{i,j}; 0) \leq B'_k - b'_{j,k,o'} \qquad \forall A_j \in A, \forall o' \in \{1, \ldots, M_j\}, \forall R_k \in \{R_1, \ldots, R_v\} \tag{8.26}$$

The last constraints (8.26) state that, for every activity A_j with mode o', and for every resource R_k, the sum of the resource demands $b'_{i,k,o}$ for R_k from the activities A_i

with mode o that overlap with A_j, should not exceed the capacity of R_k less the demand $b'_{j,k,o'}$ for R'_k from A_j.

This encoding involves $2n(n-1)$ $n(n-1)$ –i.e. $\mathcal{O}(n^2)$– new Boolean and integer variables, respectively, and $3n(n-1) + \sum_{A_i \in A} vM_i$ –i.e. $\mathcal{O}(n^2 + nvM_{max})$, where $M_{max} = max_{A_i \in A}(M_i)$– new constraints.

As in the standard RCPSP, we have the following redundant constraints encoding anti-symmetry and transitivity of the precedence relation:

$$z^1_{i,j} \vee z^1_{j,i} \qquad \forall A_i, A_j \in A, i \neq j \tag{8.27}$$

$$(z^1_{i,j} \wedge z^1_{j,k}) \rightarrow z^1_{i,k} \qquad \forall A_i, A_j, A_k \in A, i \neq j, j \neq k, i \neq k \tag{8.28}$$

The following constraints have also shown to slightly improve propagation in this encoding:

$$z^1_{i,j} \vee z^2_{i,j} \qquad \forall A_i, A_j \in A, i \neq j \tag{8.29}$$

$$z^2_{i,j} \vee z^2_{j,i} \qquad \forall A_i, A_j \in A, i \neq j \tag{8.30}$$

$$z^1_{i,j} \rightarrow z^2_{j,i} \qquad \forall A_i, A_j \in A, i \neq j \tag{8.31}$$

8.2.3 New Boolean Encoding

We have implemented an additional encoding for the MRCPSP. In this encoding a schedule of modes is a vector m of vectors of Booleans, where $m_{i,o}$ is $true$ if activity A_i acts in mode o and $false$ otherwise. Obviously, there must be one, and only one, $true$ value in each vector m_i. This property of the vectors is the **Exactly-One** constraint. This constraint is commonly expressed as the conjunction of the ALO (At-Least-One), and the AMO (At-Most-One) constraints. We have used the standard encoding of the ALO and AMO constraints described in Subsection 3.5.1. The resulting formulation is:

$$m_{0,1} \wedge m_{n+1,1} \tag{8.32}$$

$$m_{i,1} \vee \ldots \vee m_{i,M_i} \qquad \forall A_i \in A \tag{8.33}$$

$$\bigwedge_{o=1}^{M_i} \bigwedge_{j=o+1}^{M_i} \neg m_{i,o} \vee \neg m_{i,j} \qquad \forall A_i \in A \tag{8.34}$$

In this new encoding, constraints (8.12) and (8.13) disappear, and $SM_i = o$ must be replaced by $m_{i,o}$ in the remaining constraints. We have called TimeBool and TaskBool the two new formulations resulting from applying this encoding in the Time and Task formulations, respectively.

8.2.4 Experiments

The first experiment in the MRCPSP is to check if the new preprocessing method, called *non-renewable resource demand reduction*, improves the performance of the system. We consider the multimode j20 testset from the PSPLib. The comparison is made with and without this new method in the Time and Task formulations. We have run the experiments on an Intel® Core™ i5 CPU at 2.66 GHz, with 6GB of RAM, running 64-bit openSUSE 11.2 (kernel 2.6.31), using Yices 1.0.29. The timeout was set to 500 seconds for all experiments. In Table 8.2 we can see that the non-renewable resource demand reduction allows us to obtain much better results in the Task formulation; with this configuration the system solves the same number of instances than when not using this preprocess, but spending 22% less time. In the Time formulation, it also solves the same number of instances but the time saved is not as significant, only 12%.

	j20 - 554 inst.			
	Time		Task	
	With	Without	With	Without
Solution	551	551	554	554
Average	28.75	32.23	3.44	4.20
Median	7.79	8.09	0.58	0.69

Table 8.2: Comparison with and without using the non-renewable resources demand reduction preprocessing method (cutoff 500 seconds).

The second experiment is the comparison of the new TimeBool and TaskBool encodings. The experiments are performed on the following MRCPSP testsets from the PSPLib: c15, c21, j10, j12, j14, j16, j18, j20, j30, m1, m2, m4, m5, n0, n1, n3, r1, r3, r4 and r5. We have run the experiments on an Intel® Core™ i5 CPU at 2.66 GHz, with 6GB of RAM, running 64-bit openSUSE 11.2 (kernel 2.6.31), using Yices 1.0.29. The timeout was set to 500 seconds for all experiments.

In Tables 8.3 and 8.4 we can see that the task based encodings perform better than the time based encondings. The first solve 48 instances more, and the instances are solved in almost an order of magnitude less time. We also can see that the new Boolean encodings perform slightly better than the ones using integer variables. The Task formulation solves the same number of instances than the TaskBool formulation in almost equal time. But the TimeBool formulation solves 3 instances more that the Time formulation in practically identical time.

We cannot compare our approach with another solver because we have not found any one available. We have compared it with existing results in the PSPLib. The time of the j30 testsets is not available in the PSPLib. The results of the j18 and j20 testsets

seems to be wrong since they are identical, therefore we have not considered them either. In Table 8.5 we can see that our system gives better times than the ones in the PSPLib. In many testsets our system is an order of magnitude faster.

8.3 Summary

In this chapter we have shown that SMT is a competitive approach for other scheduling problems, namely the RCPSP/max and the MRCPCP . Using an algorithm with preprocessing and an appropriate encoding (the new Boolean encoding) the performances obtained were similar, and in some cases better, than the ones of the best state-of-the-art solvers. We remark that we have obtained very good performance in the MRCPCP. However, we have not been able to compare our system with other solvers. We have made our comparisons using the results in the PSPLib.

Formulation	Set	#Instances	#Solved	#Non Sol.	Mean	Median
Time	c15	551	551	0	8.92	2.75
	c21	552	552	0	9.53	3.04
	j10	536	536	0	0.98	0.44
	j12	547	547	0	2.08	0.90
	j14	551	551	0	4.17	1.60
	j16	550	550	0	8.43	2.95
	j18	552	552	0	16.04	4.98
	j20	554	551	3	28.73	7.79
	j30	640	640	78	71.44	34.83
	m1	640	462	0	0.31	0.29
	m2	481	481	0	2.43	1.06
	m4	555	555	0	20.69	5.98
	m5	558	540	18	40.31	13.19
	n0	470	470	0	10.23	2.25
	n1	637	637	0	8.15	2.27
	n3	600	600	0	7.78	2.61
	r1	553	553	0	2.42	0.99
	r3	557	557	0	15.50	4.80
	r4	552	552	0	23.16	7.77
	r5	546	546	0	32.75	12.85
	Tot.	11182	11083	99	15.66	3.04
TimeBool	c15	551	551	0	9.19	3.00
	c21	552	552	0	9.86	3.31
	j10	536	536	0	1.01	0.46
	j12	547	547	0	2.14	0.89
	j14	551	551	0	4.69	1.88
	j16	550	550	0	8.93	3.17
	j18	552	552	0	16.94	5.19
	j20	554	552	2	29.01	7.97
	j30	640	563	77	68.41	32.12
	m1	640	640	0	0.33	0.31
	m2	481	481	0	2.54	1.11
	m4	555	555	0	21.76	6.14
	m5	558	541	17	42.30	13.70
	n0	470	470	0	9.91	2.03
	n1	637	637	0	8.67	2.75
	n3	600	600	0	8.37	2.78
	r1	553	553	0	2.51	0.92
	r3	557	557	0	15.75	5.32
	r4	552	552	0	23.20	7.88
	r5	546	546	0	38.85	12.74
	Tot.	11182	11086	96	15.98	3.12

Table 8.3: Comparison between Time and TimeBool encoding (cutoff 500 seconds).

Formulation	Set	#Instances	#Solved	#Non Sol.	Mean	Median
Task	c15	551	551	0	1.15	0.26
	c21	552	552	0	0.29	0.13
	j10	536	536	0	0.05	0.04
	j12	547	547	0	0.08	0.06
	j14	551	551	0	0.19	0.11
	j16	550	550	0	0.41	0.18
	j18	552	552	0	1.06	0.34
	j20	554	554	0	3.44	0.58
	j30	640	595	45	17.62	6.31
	m1	640	640	0	0.06	0.04
	m2	481	481	0	0.16	0.10
	m4	555	555	0	1.04	0.35
	m5	558	555	3	5.06	0.69
	n0	470	470	0	0.76	0.14
	n1	637	637	0	0.40	0.18
	n3	600	600	0	0.44	0.18
	r1	553	553	0	0.22	0.12
	r3	557	557	0	0.61	0.24
	r4	552	552	0	1.12	0.31
	r5	546	546	0	1.63	0.38
	Tot.	11182	11134	48	1.84	0.20
TaskBool	c15	551	551	0	0.93	0.25
	c21	552	552	0	0.29	0.13
	j10	536	536	0	0.05	0.04
	j12	547	547	0	0.08	0.05
	j14	551	551	0	0.19	0.10
	j16	550	550	0	0.40	0.19
	j18	552	552	0	1.08	0.33
	j20	554	554	0	3.12	0.61
	j30	640	595	45	15.96	6.07
	m1	640	640	0	0.06	0.05
	m2	481	481	0	0.18	0.11
	m4	555	555	0	1.20	0.34
	m5	558	555	3	6.82	0.59
	n0	470	470	0	0.79	0.15
	n1	637	637	0	0.41	0.18
	n3	600	600	0	0.43	0.19
	r1	553	553	0	0.23	0.13
	r3	557	557	0	0.60	0.25
	r4	552	552	0	1.09	0.31
	r5	546	546	0	1.53	0.38
	Tot.	11182	11134	48	1.82	0.20

Table 8.4: Comparison between Task and TaskBool encoding (cutoff 500 seconds).

	TaskBool		PSPLib	
Set	Mean	Median	Mean	Median
c15	0.93	0.25	25.32	1.74
c21	0.29	0.13	3.20	0.56
j10	0.05	0.04	0.12	0.06
j12	0.08	0.05	0.32	0.07
j14	0.19	0.10	1.62	0.22
j16	0.40	0.19	6.37	0.38
m1	0.06	0.05	0.05	0.03
m2	0.18	0.11	1.02	0.22
m4	1.20	0.34	46.10	2.51
m5	6.82	0.59	254.90	5.47
n0	0.79	0.15	1.98	0.06
n1	0.41	0.18	3.53	0.47
n3	0.43	0.19	16.81	1.39
r1	0.23	0.13	4.84	0.72
r3	0.60	0.25	10.79	1.47
r4	1.09	0.31	10.97	1.58
r5	1.53	0.38	14.70	1.83

Table 8.5: Comparison between our approach and the PSPLib results. The number of instances in the PSPLib files and the number of results is not always consistent, so this table should only be considered as indicative.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

In this thesis, we have achieved the four objectives proposed in the introduction:

- **To show that SMT can be a good generic solving approach for CSP**. We have developed `fzn2smt`, a tool for encoding all FLATZINC instances into SMT. The good results obtained by `fzn2smt` on the MINIZINC benchmarks suggest that the SMT technology can be effectively used for solving CSPs in a broad sense. We think that `fzn2smt` can help getting a picture of the suitability of SMT solvers for solving CSPs. We also remark that the `fzn2smt` system has obtained very good results in the MINIZINC Challenges of 2010 and 2011 (a gold medal and 3 silver medals).

- **To prove that using an SMT solver in conjunction with appropriate algorithms can be a robust approach for optimization variants of CSP**. `fzn2smt` is able to solve not only decision problems, but optimization ones. In spite of the lack of support for optimization of most SMT solvers, surprisingly good results have been obtained on many optimization problems by means of successive calls to the decision procedure, performing either linear, binary or hybrid search. We have also introduced a new system, called `WSimply`, which fills the gap between CSP and SMT regarding over-constrained problems.

- **To develop a system supporting meta-constraints, allowing the user to model Weighted CSP intensionally, and to solve them using SMT**. The system `WSimply` allows intensional modelling of WCSP. It also supports some of the best-known meta-constraints from the literature and other ones that allow the user to specify richer models. In particular, we have applied these meta-constraints on two well-

known problems, the NRP and a variant of the BACP, showing how to improve the quality of the solutions and even the solving time for some cases. The usage of SMT solvers in our solving strategies is a promising choice, since several constraints, once described intensionally, can be potentially more efficiently handled. We have also implemented the SMT adaptation of the WPM1 algorithm, obtaining good performance.

- **To provide an SMT based system being competitive with state-of-the-art methods for scheduling problems**. We have implemented efficient preprocessing and solving algorithms that, together with different encodings for the RCPSP, the RCPSP/max and the MRCPSP, achieve performances comparable with the ones of the best existing approaches.

The contributions of this thesis are outlined in the introduction, but we want to highlight the *encoding of the main variants of the CP paradigm (CSP, COP and WCSP) into SMT* and the good performances obtained. We have developed the first relatively general and systematic system translating CSPs into SMT. We have developed the first system for specification and solving of intensional WCSPs with meta-constraints. We have performed tests showing that encoding the RCPSP in SMT is a good solving approximation for this problem and its generalizations. We have created a new formulation of the event encoding of the RCPSP. In the case of MRCSP we have obtained the best known performance.

On the other hand, we also want to remark some intuitive ideas confirmed during the development of this thesis:

- The higher is the Boolean component of the problems to solve, the better is the performance of SMT solvers with respect to the performance of other state-of-the-art solvers.
- The usage of SMT solvers as black-boxes to solve CSPs turns into almost unpredictable the impact of adding propagation constraints to the models.
- The usage of meta-constraints in over-constrained problems allows us to obtain better, e.g., more fair, solutions.
- Borrowing UNSAT core based optimization algorithms from the MaxSAT field to solve WSMT seems to be a promising option.
- The SMT language is very well suited for specifying scheduling problems, where there are lots of Boolean constraints. Moreover, SMT solvers and Lazy_fd, using SAT based approaches, are the best systems for solving the RCPSP and its generalizations.

9.2 Future Work

The efficiency of a solver is strongly dependent on its predefined strategies. Hence changing these heuristics could dramatically affect the SMT solver performance in other problem domains. We believe that there is much room for improvement in solving CSPs with SMT. For instance, apart from the possibility of controlling the solver strategies, we think that developing theory solvers for global constraints is also a promising research line. In fact, there exist already some interesting results in this direction. On the other hand, we think that better results could be obtained if directly translating from the MiniZinc language to SMT and avoiding some of the flattening. In doing so, most clever translations for the SMT solvers could be possible and probably less variables could be generated. For instance, MiniZinc disjunctions of arithmetic constraints are translated into FlatZinc constraints by reifying them with auxiliary Boolean variables. In our approach this step is not needed since it is already done by the SMT solver.

A better approach to optimization in SMT is also a pending issue. So we might try to use the theory of costs or new decision based algorithms appeared in the last year for the SAT setting.

With respect to `WSimply` we also plan to extend our framework with support for Integer Programming techniques, which may be more suitable for some problems where the Boolean structure is less important than the arithmetic expressions. We want to remark that we have also proposed a similar extension to deal with weighted CSPs in MiniZinc, to which we could easily provide a solving mechanism based on SMT and MaxSMT as is done for `WSimply`.

Finally, for the scheduling problems, we plan to develop new encodings based on events, and study the channeling of different encodings to achieve greater propagation. We also want to solve, using SMT, other scheduling problems such as: Job-shop, Multi-Mode Resource Constrained Project Scheduling Problem with Minimal and Maximal Time Lags (MRCPSP/max), Multi-Skill Project Scheduling Problem (MSPSP), Resource Investment Problem with Minimal and Maximal Time Lags (RIP/max), etc.

Bibliography

[ABL09] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy, *Solving (Weighted) Partial MaxSAT through Satisfiability Testing*, Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, LNCS, vol. 5584, Springer, 2009, pp. 427–440.

[ABP+11a] Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *A Proposal for Solving Weighted CSPs with SMT*, Proceedings of the 10th International Workshop on Constraint Modelling and Reformulation (ModRef 2011), 2011, pp. 5–19.

[ABP+11b] Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Satisfiability Modulo Theories: An Efficient Approach for the Resource-Constrained Project Scheduling Problem*, SARA (Michael R. Genesereth and Peter Z. Revesz, eds.), AAAI, 2011, pp. 2–9.

[ABP+11c] Carlos Ansótegui, Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *W-MiniZinc: A Proposal for Modeling Weighted CSPs with MiniZinc*, Proceedings of the first Minizinc workshop (MZN-2011), 2011.

[Ack68] W. Ackermann, *Solvable cases of the decision problem*, Studies in logic and the foundations of mathematics, North-Holland, 1968.

[ADN07] Christian Artigues, Sophie Demassey, and Emmanuel Neron, *Resource-constrained project scheduling: Models, algorithms, extensions and applications*, ISTE, 2007.

[AM04] Carlos Ansótegui and Felip Manyà, *Mapping Problems with Finite-Domain Variables to Problems with Boolean Variables*, SAT (Selected Papers (Holger H. Hoos and David G. Mitchell, eds.), Lecture Notes in Computer Science, vol. 3542, Springer, 2004, pp. 1–15.

[AMR03] Christian Artigues, Philippe Michelon, and Stephane Reusser, *Insertion Techniques for Static and Dynamic Resource-Constrained Project Scheduling*, European Journal of Operational Research **149** (2003), no. 2, 249–267.

[AW07] Krzysztof R. Apt and Mark Wallace, *Constraint Logic Programming using Eclipse*, Cambridge University Press, New York, NY, USA, 2007.

[Bap09] Philippe Baptiste, *Constraint-Based Schedulers, Do They Really Work?*, Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming, LNCS, vol. 5732, Springer, 2009, pp. 1–1.

[Bar05] Roman Barták, *Constraint propagation and backtracking-based search*, First international summer school on CP, Maratea, Italy, 2005.

[BB82] D.D. Bedworth and J.E. Bailey, *Integrated production control systems: management, analysis, design*, Wiley, 1982.

[BB03] Olivier Bailleux and Yacine Boufkhad, *Efficient CNF Encoding of Boolean Cardinality Constraints*, Principles and Practice of Constraint Programming, CP 2003 (Francesca Rossi, ed.), Lecture Notes in Computer Science, vol. 2833, Springer Berlin / Heidelberg, 2003, pp. 108–122.

[BBC+06] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani, *Efficient Theory Combination via Boolean Search*, Information and Computation **204** (2006), no. 10, 1493–1525.

[BCBL04] Edmund K. Burke, Patrick De Causmaecker, Greet Vanden Berghe, and Hendrik Van Landeghem, *The State of the Art of Nurse Rostering*, J. Scheduling **7** (2004), no. 6, 441–499.

[BCF+06] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Alessandro Santuari, and Roberto Sebastiani, *To Ackermann-ize or Not to Ackermann-ize? On Efficiently Handling Uninterpreted Function Symbols in $SMT(\mathcal{EUF} \cup \mathcal{T})$*, in Hermann and Voronkov [HV06], pp. 557–571.

[BCF+08] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani, *The MathSAT 4 SMT Solver*, in *CAV* [DBL08], pp. 299–303.

[BDM+99] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch, *Resource-Constrained Project Scheduling: Notation, Classification, Models, and Methods*, European Journal of Operational Research **112** (1999), no. 1, 3 – 41.

[Bee12] *BEE: Equi-Propagation Encoder*, http://amit.metodi.me/research/bee/, 2012.

[Bes06] Christian Bessiere, *Constraint propagation*, Tech. report, In, 2006.

[BFM97] Thomas Back, David B. Fogel, and Zbigniew Michalewicz (eds.), *Handbook of Evolutionary Computation*, 1st ed., IOP Publishing Ltd., Bristol, UK, UK, 1997.

[BG06] Armin Biere and Carla P. Gomes (eds.), *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, Lecture Notes in Computer Science, vol. 4121, Springer, 2006.

[BGS99] Laure Brisoux, Éric Grégoire, and Lakhdar Sais, *Improving Backtrack Search for SAT by Means of Redundancy*, ISMIS (Zbigniew W. Ras and Andrzej Skowron, eds.), Lecture Notes in Computer Science, vol. 1609, Springer, 1999, pp. 301–309.

[BHM01] Ramón Béjar, Reiner Hähnle, and Felip Manyà, *A Modular Reduction of Regular Logic to Classical Logic*, ISMVL, 2001, pp. 221–226.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (eds.), *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications, vol. 185, IOS Press, 2009.

[BKO+07] Randal E. Bryant, Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady, *Deciding Bit-Vector Arithmetic with Abstraction*, TACAS (Orna Grumberg and Michael Huth, eds.), Lecture Notes in Computer Science, vol. 4424, Springer, 2007, pp. 358–372.

[BLK83] J. Blazewicz, J. K. Lenstra, and A. H. G. Rinnooy Kan, *Scheduling Subject to Resource Constraints: Classification and Complexity*, Discrete Applied Mathematics **5** (1983), no. 1, 11 – 24.

[BLO+12] Cristina Borralleras, Salvador Lucas, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio, *SAT Modulo Linear Arithmetic for Solving Polynomial Constraints*, J. Autom. Reasoning **48** (2012), no. 1, 107–131.

[BM10] Milan Bankovic and Filip Maric, *An Alldifferent Constraint Solver in SMT*, Proceedings of the 2010 SMT Workshop, 2010.

[BMR88] M. Bartusch, R. H. Mohring, and F. J. Radermacher, *Scheduling Project Networks with Resource Constraints and Time Windows*, Annals of Operations Research **16** (1988), 201–240.

[BMR97] Stefano Bistarelli, Ugo Montanari, and Francesca Rossi, *Semiring-based constraint satisfaction and optimization*, J. ACM **44** (1997), no. 2, 201–236.

[BMR+99] S. Bistarelli, U. Montanari, F. Rossi, T. Schiex, G. Verfaillie, and H. Fargier, *Semiring-Based CSPs and valued CSPs: Frameworks, Properties, and Comparison*, Constraints **4** (1999), no. 3, 199–240.

[BNO+08a] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio, *A Write-Based Solver for SAT Modulo the Theory of Arrays*, FMCAD (Alessandro Cimatti and Robert B. Jones, eds.), IEEE, 2008, pp. 1–8.

[BNO+08b] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio, *The Barcelogic SMT Solver*, in *CAV* [DBL08], pp. 294–298.

[BPSV09] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *SIMPLY: a Compiler from a CSP Modeling Language to the SMT-LIB Format*, Proceedings of the 8th International Workshop on Constraint Modelling and Reformulation, 2009, pp. 30–44.

[BPSV12] Miquel Bofill, Miquel Palahí, Josep Suy, and Mateu Villaret, *Solving constraint satisfaction problems with SAT modulo theories*, Constraints **17** (2012), no. 3, 273–303.

[BS94] Belaid Benhamou and Lakhdar Sais, *Tractability Through Symmetries in Propositional Calculus*, J. Autom. Reasoning **12** (1994), no. 1, 89–102.

[BST10a] Clark Barrett, Aaron Stump, and Cesare Tinelli, *The Satisfiability Modulo Theories Library (SMT-LIB)*, http://www.SMT-LIB.org, 2010.

[BST10b] Clark Barrett, Aaron Stump, and Cesare Tinelli, *The SMT-LIB Standard: Version 2.0*, Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK) (A. Gupta and D. Kroening, eds.), 2010.

[BSV10] Miquel Bofill, Josep Suy, and Mateu Villaret, *A System for Solving Constraint Satisfaction Problems with SMT*, Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing, LNCS, vol. 6175, Springer, 2010, pp. 300–305.

[Cab11] Alba Cabiscol, *Encodings and Benchmarks for MaxSAT Solving*, Universitat de Lleida, Departament d'Informàtica i Enginyeria Industrial, Lleida, Spain, 2011.

[CB94] James M. Crawford and Andrew B. Baker, *Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems*, In Proceedings of the Twelfth National Conference on Artificial Intelligence, 1994, pp. 1092–1097.

[CFG+10] Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani, and Cristian Stenico, *Satisfiability Modulo the Theory of Costs: Foundations and Applications*, TACAS, LNCS, vol. 6015, 2010, pp. 99–113.

[CIP+00] Marco Cadoli, Giovambattista Ianni, Luigi Palopoli, Andrea Schaerf, and Domenico Vasile, *NP-SPEC: an executable specification language for solving all problems in NP*, Computer Languages **26** (2000), no. 2–4, 165–195.

[CM01] Carlos Castro and Sebastian Manzano, *Variable and Value Ordering When Solving Balanced Academic Curriculum Problems*, 6th Annual Workshop of the ERCIM Working Group on Constraints, 2001.

[CMP06] Marco Cadoli, Toni Mancini, and Fabio Patrizi, *SAT as an Effective Solving Technology for Constraint Problems*, Proceedings of the 16th International Symposium on Foundations of Intelligent Systems, LNCS, vol. 4203, Springer, 2006, pp. 540–549.

[CN03] Jacques Carlier and Emmanuel Néron, *On Linear Lower Bounds for the Resource Constrained Project Scheduling Problem*, European Journal of Operational Research **149** (2003), no. 2, 314–324.

[CS00] Philippe Chatalic and Laurent Simon, *ZRes: The old Davis-Putnam procedure meets ZBDD*, In 17th Intl. Conf. on Automated Deduction (CADE17), volume 1831 of LNAI, Springer-Verlag, 2000, pp. 449–454.

[CS05] Marco Cadoli and Andrea Schaerf, *Compiling Problem Specifications into SAT*, Artificial Intelligence **162** (2005), no. 1–2, 89–120.

[CV11] José Coelho and Mario Vanhoucke, *Multi-mode resource-constrained project scheduling using RCPSP and SAT solvers.*, European Journal of Operational Research **213** (2011), no. 1, 73–82.

[DBL01] *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, ACM, 2001.

[DBL08] *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, LNCS, vol. 5123, 2008.

[DBL09] *Principles and practice of constraint programming - cp 2009, 15th international conference, cp 2009, lisbon, portugal, september 20-24, 2009, proceedings*, LNCS, vol. 5732, Springer, 2009.

[DdM06a] Bruno Dutertre and Leonardo Mendonça de Moura, *A Fast Linear-Arithmetic Solver for DPLL(T)*, CAV (Thomas Ball and Robert B. Jones, eds.), Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 81–94.

[DdM06b] Bruno Dutertre and Leonardo Mendonça de Moura, *The Yices SMT solver*, Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[Dec90] Rina Dechter, *On the expressiveness of networks with hidden variables*, Proceedings of the eighth National conference on Artificial intelligence - Volume 1, AAAI'90, AAAI Press, 1990, pp. 556–562.

[DF98] Rina Dechter and Daniel Frost, *Backtracking algorithms for constraint satisfaction problems - a tutorial survey*, Tech. report, 1998.

[DGSA] Simon De Givry, Thomas Schiex, and David Allouche, *Toulbar2*, http://mulcyber.toulouse.inra.fr/projects/toulbar2/.

[DHN06] Nachum Dershowitz, Ziyad Hanna, and Er Nadel, *A scalable algorithm for minimal unsatisfiable core extraction*, In Proc. SAT06, Springer, 2006.

[DLL62] Martin Davis, George Logemann, and Donald Loveland, *A machine program for theorem-proving*, Communications of the ACM **5** (1962), no. 7, 394–397.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner, *Z3: An Efficient SMT Solver*, TACAS, LNCS, vol. 4963, 2008, pp. 337–340.

[dMR04] L. de Moura and H. Ruess, *An Experimental Evaluation of Ground Decision Procedures*, 16th International Conference on Computer Aided Verification, CAV'04 (R. Alur and D. Peled, eds.), Lecture Notes in Computer Science, vol. 3114, Springer, 2004, pp. 162–174.

[DP60] Martin Davis and Hilary Putnam, *A Computing Procedure for Quantification Theory*, Journal of the ACM **7** (1960), 201–215.

[DP89] Rina Dechter and Judea Pearl, *Tree clustering for constraint networks (research note)*, Artif. Intell. **38** (1989), no. 3, 353–366.

[DPBC93] Olivier Dubois, André Pascal, Yacine Boufkhad, and Jaques Carlier, *Can a very simple algorithm be eficient for solving SAT problem?*, Proc. of the DIMACS Challenge II Workshop, 1993.

[DPPH00] U. Dorndorf, E. Pesch, and T. Phan-Huy, *A Branch-and-Bound Algorithm for the Resource-Constrained Project Scheduling Problem*, Mathematical Methods of Operations Research **52** (2000), 413–439.

[DV07] Dieter Debels and Mario Vanhoucke, *A Decomposition-Based Genetic Algorithm for the Resource-Constrained Project-Scheduling Problem*, Operations Research **55** (2007), no. 3, 457–469.

[FHJ+08] A. Frisch, W. Harvey, C. Jefferson, B. Martínez-Hernández, and I. Miguel, *Essence: A Constraint Language for Specifying Combinatorial Problems*, Constraints **13** (2008), no. 3, 268–306.

[FM06] Zhaohui Fu and Sharad Malik, *On Solving the Partial MAX-SAT Problem*, in Biere and Gomes [BG06], pp. 252–265.

[FP01] Alan M. Frisch and Timothy J. Peugniez, *Solving Non-Boolean Satisfiability Problems with Stochastic Local Search*, IJCAI (Bernhard Nebel, ed.), Morgan Kaufmann, 2001, pp. 282–290.

[Fre78] Eugene C. Freuder, *Synthesizing constraint expressions*, Commun. ACM **21** (1978), no. 11, 958–966.

[Fre95] Jon William Freeman, *Improvements To Propositional Satisfiability Search Algorithms*, 1995.

[FS09] Thibaut Feydy and Peter J. Stuckey, *Lazy Clause Generation Reengineered*, in *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming* [DBL09], pp. 352–366.

[FW92] Eugene C. Freuder and Richard J. Wallace, *Partial Constraint Satisfaction*, Artif. Intell. **58** (1992), no. 1-3, 21–70.

[Fzn08] *FznTini*, http://www.sat4j.org/howto.php, 2008.

[G1210] *Minizinc + Flatzinc*, http://www.g12.csse.unimelb.edu.au/minizinc/, 2010.

[Gas79] J. Gaschnig, *Performance Measurement and Analysis of Certain Search Algorithms*, Ph.D. thesis, Carnegie-Mellon University, 1979.

[Glo89] Fred Glover, *Tabu Search - Part I*, INFORMS Journal on Computing **1** (1989), no. 3, 190–206.

[Glo90] Fred Glover, *Tabu Search - Part II*, INFORMS Journal on Computing **2** (1990), no. 1, 4–32.

[Gom58] R. E. Gomory, *Outline of an Algorithm for Integer Solutions to Linear Programs*, Bulletin of the American Society **64** (1958), 275–278.

[HB10] Sönke Hartmann and Dirk Briskorn, *A Survey of Variants and Extensions of the Resource-Constrained Project Scheduling Problem*, European Journal of Operational Research **207** (2010), no. 1, 1 – 14.

[HE80] R. Haralick and G. Elliot, *Increasing Tree Search Efficiency for Constraint Satisfaction Problems*, Artificial Intelligence **14** (1980), no. 3, 263–313.

[HK00] Sönke Hartmann and Rainer Kolisch, *Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem*, European Journal of Operational Research **127** (2000), no. 2, 394–407.

[HKW02] Brahim Hnich, Zeynep Kiziltan, and Toby Walsh, *Modelling a Balanced Academic Curriculum Problem*, Proceedings of the Fourth International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR 2002), 2002, pp. 121–131.

[HL05] Willy Herroelen and Roel Leus, *Project Scheduling under Uncertainty: Survey and Research Potentials*, European Journal of Operational Research **165** (2005), no. 2, 289–306.

[Hor10] Andrei Horbach, *A Boolean Satisfiability Approach to the Resource-Constrained Project Scheduling Problem*, Annals of Operations Research **181** (2010), 89–107.

[HRD98] Willy Herroelen, Bert De Reyck, and Erik Demeulemeester, *Resource-Constrained Project Scheduling: A Survey of Recent Developments*, Computers and Operations Research **25** (1998), no. 4, 279 – 302.

[Hua07] Jinbo Huang, *The Effect of Restarts on the Efficiency of Clause Learning*, IJCAI (Manuela M. Veloso, ed.), 2007, pp. 2318–2323.

[Hua08] Jinbo Huang, *Universal Booleanization of Constraint Models*, CP (Peter J. Stuckey, ed.), Lecture Notes in Computer Science, vol. 5202, Springer, 2008, pp. 144–158.

[HV06] Miki Hermann and Andrei Voronkov (eds.), *Logic for programming, artificial intelligence, and reasoning*, Lecture Notes in Computer Science, vol. 4246, Springer, 2006.

[IM94] Kazuo Iwama and Shuichi Miyazaki, *SAT-Variable Complexity of Hard Combinatorial Problems*, Proceedings of the world computer congres of the IFIP, Elsevier Science Inc., 1994, pp. 253–258.

[Jac10] *JaCoP Java Constraint Programming Solver*, http://jacop.osolpro.com, 2010.

[JW90] Robert G. Jeroslow and Jinchang Wang, *Solving Propositional Satisfiability Problems*, Ann. Math. Artif. Intell. **1** (1990), 167–187.

[KALM11] Oumar Koné, Christian Artigues, Pierre Lopez, and Marcel Mongeau, *Event-Based MILP Models for Resource-Constrained Project Scheduling Problems*, Computers & Operations Research **38** (2011), 3–13.

[Kau06] Henry A. Kautz, *Deconstructing Planning as Satisfiability*, Proceedings of the Twenty-first Conference on Artificial Intelligence, AAAI Press, 2006, pp. 1524–1526.

[KGN+09] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik, *Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation*, CAV (Ahmed Bouajjani and Oded Maler, eds.), Lecture Notes in Computer Science, vol. 5643, Springer, 2009, pp. 414–429.

[KH06] Rainer Kolisch and Sönke Hartmann, *Experimental investigation of heuristics for resource-constrained project scheduling: An update*, European Journal of Operational Research **174** (2006), no. 1, 23–37.

[Kol96] Rainer Kolisch, *Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation*, European Journal of Operational Research **90** (1996), no. 2, 320–333.

[KP01] R. Kolisch and R. Padman, *An Integrated Survey of Deterministic Project Scheduling*, Omega **29** (2001), no. 3, 249–272.

[KS97] Rainer Kolisch and Arno Sprecher, *PSPLIB - A Project Scheduling Problem Library*, European Journal of Operational Research **96** (1997), no. 1, 205–216.

[KS99] Robert Klein and Armin Scholl, *Computing Lower Bounds by Destructive Improvement: An Application to Resource-Constrained Project Scheduling*, European Journal of Operational Research **112** (1999), no. 2, 322–346.

[KSHK07] Daher Kaiss, Marcelo Skaba, Ziyad Hanna, and Zurab Khasidashvili, *Industrial Strength SAT-based Alignability Algorithm for Hardware Equivalence Verification*, FMCAD, IEEE Computer Society, 2007, pp. 20–26.

[Kum92] Vipin Kumar, *Algorithms for Constraint Satisfaction Problems: A Survey*, AI Magazine **13** (1992), no. 1, 32–44.

[LM08] Olivier Liess and Philippe Michelon, *A Constraint Programming Approach for the Resource-Constrained Project Scheduling Problem*, Annals of Operations Research **157** (2008), 25–36.

[LMS05] Inês Lynce and João Marques-Silva, *Efficient Data Structures for Backtrack Search SAT Solvers*, Annals of Mathematics and Artificial Intelligence **43** (2005), no. 1-4, 137–152.

[LMS06] Inês Lynce and João Marques-Silva, *Efficient Haplotype Inference with Boolean Satisfiability*, AAAI, AAAI Press, 2006, pp. 104–109.

[LS04] Javier Larrosa and Thomas Schiex, *Solving weighted CSP by maintaining arc consistency*, Artif. Intell. **159** (2004), no. 1-2, 1–26.

[Mac77] Alan Mackworth, *Consistency in Networks of Relations*, Artificial Intelligence **8** (1977), no. 1, 99–118.

[MBL09] Jean-Philippe Métivier, Patrice Boizumault, and Samir Loudni, *Solving Nurse Rostering Problems Using Soft Global Constraints*, CP 2009, LNCS, vol. 5732, 2009, pp. 73–87.

[MC12] Amit Metodi and Michael Codish, *Compiling Finite Domain Constraints to SAT with BEE*, CoRR **abs/1206.3883** (2012).

[MH86] Roger Mohr and Thomas C. Henderson, *Arc and Path Consistency Revisited*, Artif. Intell. **28** (1986), no. 2, 225–233.

[MJ10] Filip Maric and Predrag Janicic, *URBiVA: Uniform Reduction to Bit-Vector Arithmetic*, IJCAR (Jürgen Giesl and Reiner Hähnle, eds.), Lecture Notes in Computer Science, vol. 6173, Springer, 2010, pp. 346–352.

[MMRB98] Aristide Mingozzi, Vittorio Maniezzo, Salvatore Ricciardelli, and Lucio Bianco, *An Exact Algorithm for the Resource-Constrained Project Scheduling Problem Based on a New Mathematical Formulation*, Management Science **44** (1998), 714–729.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, *Chaff: Engineering an Efficient SAT Solver*, in *DAC* [DBL01], pp. 530–535.

[MS01] Pedro Meseguer and Martí Sánchez, *Specializing Russian Doll Search*, In Proceedings of CP, Springer Verlag, 2001, pp. 464–478.

[MSP09] Vasco M. Manquinho, João P. Marques Silva, and Jordi Planes, *Algorithms for Weighted Boolean Optimization*, Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing, LNCS, vol. 5584, Springer, 2009, pp. 495–508.

[MTW+99] Patrick Mills, Edward Tsang, Richard Williams, John Ford, James Borrett, and Wivenhoe Park, *EaCL 1.5: An Easy Constraint optimisation Programming Language*, Tech. Report CSM-324, University of Essex, Colchester, U.K., 1999.

[Nic07] Nicholas Nethercote and Peter J. Stuckey and Ralph Becket and Sebastian Brand and Gregory J. Duck and Guido Tack, *MiniZinc: Towards a Standard CP Modelling Language*, CP, LNCS, vol. 4741, 2007, pp. 529–543.

[NO79] G. Nelson and D. C. Oppen, *Simplification by Cooperating Decision Procedures*, ACM Transactions on Programming Languages and Systems, TOPLAS **1** (1979), no. 2, 245–257.

[NO80] Greg Nelson and Derek C. Oppen, *Fast decision procedures based on congruence closure*, J. ACM **27** (1980), no. 2, 356–364.

[NO05] Robert Nieuwenhuis and Albert Oliveras, *DPLL(T) with exhaustive theory propagation and its application to difference logic*, In CAV'05 LNCS 3576, Springer, 2005, pp. 321–334.

[NO06] Robert Nieuwenhuis and Albert Oliveras, *On SAT Modulo Theories and Optimization Problems*, in Biere and Gomes [BG06], pp. 156–169.

[NORCR07] Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio, *Challenges in Satisfiability Modulo Theories*, Proceedings of the 18th International Conference on Term Rewriting and Applications, LNCS, vol. 4533, Springer, 2007, pp. 2–18.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli, *Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T)*, Journal of the ACM **53** (2006), no. 6, 937–977.

[NSB+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack, *MiniZinc: Towards a Standard CP Modelling Language*, CP 2007, LNCS, vol. 4741, 2007, pp. 529–543.

[OEK99] A. O. El-Kholy, *Resource Feasibility in Planning*, Ph.D. thesis, Imperial College, University of London, 1999.

[OSC09] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish, *Propagation via lazy clause generation*, Constraints **14** (2009), no. 3, 357–391.

[PdMB10] R. Piskac, Leonardo Mendonça de Moura, and N. Bjørner, *Deciding Effectively Propositional Logic using DPLL and substitution sets*, Journal of Automated Reasoning **44** (2010), no. 4, 401–424.

[PR91] Manfred Padberg and Giovanni Rinaldi, *A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems*, SIAM Rev. **33** (1991), no. 1, 60–100.

[PRB00] T. Petit, J. C. Regin, and C. Bessiere, *Meta-constraints on violations for over constrained problems*, ICTAI 2000, Proceedings of the 12th IEEE International Conference on Tools with Artificial Intelligence, 2000, pp. 358–365.

[Pre93] Daniele Pretolani, *Efficiency and stability of hypergraph SAT algorithms*, Proc. of the DIMACS Challenge II Workshop, 1993.

[PW96] Lawrence J. Watters Pritsker, A. Alan B. and Philip S. Wolfe, *Multiproject Scheduling with Limited Resources: A Zero-One Programming Approach*, Management Science **16** (1996), 93–108.

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, *Handbook of constraint programming (foundations of artificial intelligence)*, Elsevier Science Inc., New York, NY, USA, 2006.

[RL09] Olivier Roussel and Christophe Lecoutre, *XML Representation of Constraint Networks: Format XCSP 2.1*, CoRR **abs/0902.2362** (2009).

[Rob65] J. A. Robinson, *A machine-oriented logic based on the resolution principle*, J. ACM **12** (1965), no. 1, 23–41.

[RPD90] Francesca Rossi, Charles Petrie, and Vasant Dhar, *On the equivalence of constraint satisfaction problems*, In Proceedings of the 9th European Conference on Artificial Intelligence, 1990, pp. 550–556.

[RS08] Vadim Ryvchin and Ofer Strichman, *Local restarts*, SAT (Hans Kleine Büning and Xishun Zhao, eds.), Lecture Notes in Computer Science, vol. 4996, Springer, 2008, pp. 271–276.

[RT06] S. Ranise and C. Tinelli, *The SMT-LIB Standard: Version 1.2*, Tech. report, Dept. of Comp. Science, University of Iowa, 2006, http://www.SMT-LIB.org.

[Sat05] *Sat4j: the Boolean satisfaction and optimization library for Java*, http://users.cecs.anu.edu.au/ jinbo/fzntini/, 2005.

[SBDL01] A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt, *A Decision Procedure for an Extensional Theory of Arrays*, 16th Annual Symposium on Logic in Computer Science, IEEE Computer Society, 2001, pp. 29–37.

[SBF10] Peter J. Stuckey, Ralph Becket, and Julien Fischer, *Philosophy of the MiniZinc challenge*, Constraints **15** (2010), 307–316.

[Sch89] Uwe Schöning, *Logic for computer scientists, volume 8 of progress in computer science and applied logic*, Birkhäuser, 1989.

[Sci10] *SCIP, Solving constraint integer programs*, http://scip.zib.de/scip.shtml, 2010.

[SD98] Arno Sprecher and Andreas Drexl, *Multi-mode resource-constrained project scheduling by a simple, general and powerful sequencing algorithm*, European Journal of Operational Research **107** (1998), no. 2, 431 – 450.

[Seb07] Roberto Sebastiani, *Lazy Satisability Modulo Theories*, Journal on Satisfiability, Boolean Modeling and Computation **3** (2007), no. 3-4, 141–224.

[SFSW09] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark Wallace, *Why Cumulative Decomposition Is Not as Bad as It Sounds*, in *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming* [DBL09], pp. 746–761.

[SFSW10] Andreas Schutt, Thibaut Feydy, Peter Stuckey, and Mark Wallace, *Explaining the Cumulative Propagator*, Constraints (2010), 1–33.

[Sho84] Robert E. Shostak, *Deciding Combinations of Theories*, J. ACM **31** (1984), no. 1, 1–12.

[Sic10] *SICStus Prolog*, http://www.sics.se/sisctus, 2010.

[Sil99] João P. Marques Silva, *The Impact of Branching Heuristics in Propositional Satisfiability Algorithms*, Proceedings of the 9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence (London, UK, UK), EPIA '99, Springer-Verlag, 1999, pp. 62–74.

[Sin05] Carsten Sinz, *Towards an optimal cnf encoding of boolean cardinality constraints*, In Proc. of the 11th Intl. Conf. on Principles and Practice of Constraint Programming (CP 2005, 2005, pp. 827–831.

[SKD95] Arno Sprecher, Rainer Kolisch, and Andreas Drexl, *Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem*, European Journal of Operational Research **80** (1995), no. 1, 94 – 102.

[SLT10] C. Schulte, M. Lagerkvist, and G. Tack, *Gecode*, http://www.gecode.org, 2010.

[Spe05] *NP-SPEC Project (Spec2SAT version)*, http://www.dis.uniroma1.it/cadoli/research/projects/NP-SPEC/code/spec2SAT/, 2005.

[SS06] Hossein M. Sheini and Karem A. Sakallah, *From Propositional Satisfiability to Satisfiability Modulo Theories*, in Biere and Gomes [BG06], pp. 1–9.

[Sug11] *Sugar: A SAT-based Constraint Solver*, http://bach.istc.kobe-u.ac.jp/sugar/, 2011.

[TTKB09] Naoyuki Tamura, Akiko Taga, Satoshi Kitagawa, and Mutsunori Banbara, *Compiling Finite Linear CSP into SAT*, Constraints **14** (2009), no. 2, 254–272.

[Urq87] Alasdair Urquhart, *Hard examples for resolution*, J. ACM **34** (1987), no. 1, 209–219.

[VB01] Miroslav N. Velev and Randal E. Bryant, *Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors*, in *DAC* [DBL01], pp. 226–231.

[VLS96] Gérard Verfaillie, Michel Lemaître, and Thomas Schiex, *Russian Doll Search for Solving Constraint Optimization Problems*, AAAI/IAAI, Vol. 1 (William J. Clancey and Daniel S. Weld, eds.), AAAI Press / The MIT Press, 1996, pp. 181–187.

[VM07] M Vanhoucke and B Maenhout, *NSPLIB - a nurse scheduling problem library: A tool to evaluate (meta-) heuristic procedures*, Operational Research for Health Policy Making Better Decisions (2007), 151–165.

[Wal96] Richard J. Wallace, *Enhancements of Branch and Bound Methods for the Maximal Constraint Satisfaction Problem*, Proc. of AAAI-96, 1996, pp. 188–195.

[Wal00] Toby Walsh, *SAT v CSP*, Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, LNCS, vol. 1894, Springer, 2000, pp. 441–456.

[ZBY06] Guidong Zhu, Jonathan F. Bard, and Gang Yu, *A Branch-and-Cut Procedure for the Multimode Resource-Constrained Project-Scheduling Problem*, INFORMS J. on Computing **18** (2006), no. 3, 377–390.

[Zha97] Hantao Zhang, *SATO: An Efficient Propositional Prover*, CADE (William McCune, ed.), Lecture Notes in Computer Science, vol. 1249, Springer, 1997, pp. 272–275.

[ZHR08] Juan Camilo Zapata, Bri Mathias Hodge, and Gintaras V. Reklaitis, *The multimode resource constrained multiproject scheduling problem: Alternative formulations*, AIChE Journal **54** (2008), no. 8, 2101–2119.

[ZLS04] Hantao Zhang, Dapeng Li, and Haiou Shen, *A SAT Based Scheduler for Tournament Schedules*, Theory and Applications of Satisfiability Testing, 7th International Conference, SAT'04, Online Proceedings, 2004, pp. 191–196.

[ZLS06] Jianmin Zhang, Sikun Li, and Shengyu Shen, *Extracting minimum unsatisfiable cores with a greedy genetic algorithm*, Proceedings of the 19th Australian joint conference on Artificial Intelligence: advances in Artificial Intelligence (Berlin, Heidelberg), AI'06, Springer-Verlag, 2006, pp. 847–856.

[ZM88] Ramin Zabih and David A. McAllester, *A rearrangement search strategy for determining propositional satisfiability*, In Proceedings of teh National Conference on Artificial Intelligence AAAI, 1988, pp. 155–160.

[ZM02] Lintao Zhang and Sharad Malik, *The Quest for Efficient Boolean Satisfiability Solvers*, CAV, LNCS, vol. 2404, Springer, 2002, pp. 17–36.

[ZM03] Lintao Zhang and Sharad Malik, *Cache Performance of SAT Solvers: a Case Study for Efficient Implementation of Algorithms*, SAT (Enrico Giunchiglia and Armando Tacchella, eds.), Lecture Notes in Computer Science, vol. 2919, Springer, 2003, pp. 287–298.