UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

TESIS DOCTORAL

Mejora de la eficiencia de resolución del sistema TOY(FD) y su aplicación a problemas reales de la industria

Improving the Solving Efficiency Of TOY (FD) and its application to Real-Life Problems

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Ignacio Castiñeiras Pérez

Directores

Fernando Sáenz Pérez
Francisco Javier López Fraguas

Madrid, 2014

Mejora de la eficiencia de resolución del sistema TOY(FD) y su aplicación a problemas reales de la industria

Tesis Doctoral

Ignacio Castiñeiras Pérez

Directores

Fernando Sáenz Pérez
Francisco Javier López Fraguas

Universidad Complutense de Madrid
Facultad de Informática
Departamento de Sistemas Informáticos y Computación
Madrid, marzo de 2014

Improving the Solving Efficiency of TOY(FD) and its Application to Real-Life Problems

PhD Thesis

Ignacio Castiñeiras Pérez

Advisors

Fernando Sáenz Pérez

Francisco Javier López Fraguas

Complutense Univesity of Madrid
Faculty of Information Technology and Computer Science
Departament of Information Systems and Computing

Madrid, March of 2014

Mejora de la eficiencia de resolución del sistema TOY(FD) y su aplicación a problemas reales de la industria

Improving the Solving Efficiency of TOY(FD) and its Application to Real-Life Problems

Ignacio Castiñeiras Pérez

Tesis doctoral, con título “Mejora de la eficiencia de resolución del sistema TOY(FD) y su aplicación a problemas reales de la industria”, presentada por Ignacio Castiñeiras Pérez para la obtención del título de Doctor por la Universidad Complutense de Madrid.

Título:
Mejora de la eficiencia de resolución del sistema TOY(FD)
y su aplicación a problemas reales de la industria
Autor:
Ignacio Castiñeiras Pérez (ncasti@fdi.ucm.es)
Directores:
Fernando Sáenz Pérez (fernan@sip.ucm.es)
Francisco Javier López Fraguas (fraguas@sip.ucm.es)

Universidad Complutense de Madrid, Facultad de Informática, Departamento de Sistemas Informáticos y Computación
Madrid, marzo de 2014

PhD Thesis, with title “Improving the Solving Efficiency of TOY(FD) and its Application to Real-Life Problems”, presented by Ignacio Castiñeiras Pérez for obtaining a PhD at the Complutense University of Madrid.

Título:
Improving the Solving Efficiency of TOY(FD)
and its Application to Real-Life Problems
Author:
Ignacio Castiñeiras Pérez (ncasti@fdi.ucm.es)
Advisors:
Fernando Sáenz Pérez (fernan@sip.ucm.es)
Francisco Javier López Fraguas (fraguas@sip.ucm.es)

Complutense Univesity of Madrid, Faculty of Information Technology and Computer Science, Departament of Information Systems and Computing
Madrid, March of 2014

- As you requested, we took some fibre samples from Nordberg's jacket. Use this microscope to take a look for yourself.

- I can't see anything.

- Use your open eye, Frank.

- Yeah, I can see it now.

Lab manager Ted Olsen (Ed Williams),
captain Ed Hocken (George Kennedy)
and lieutenant Frank Drebin (Leslie Nielsen).
The Naked Gun, 1988.

Acknowledgements

The day I joined the Declarative Programming Group (GPD) for starting the master program it was my 24^{th} birthday, and the day I am finishing this PhD thesis is the 23^{rd} birthday of my sister. If I wanted to encourage her for joining a post-graduate program, I would not know what to say. Research has, as any other discipline, several advantages and drawbacks. But, if I think back, it has given me two mayor points, which I want to point out.

I started the post-graduate for the sake of becoming a teacher. During these years I have had the opportunity of helping professors on lab subjects, and I have enjoyed it a lot. But, besides any concrete concept I taught students (which I hope they understood), research has taught me to ask myself for the why of the things. That is, to zoom out from the concrete topic we were working on, and start asking myself questions like: *Why are we working on this? Are we the first ones working on this? What do the others do, what do they add to the topic? What interactions can be done from our approaches? Can we find new ways for collaborating and start new approaches together?* This valuable way of thinking will be very useful to me for the rest of my life, whatever I do for living. Thus, thanks for that.

The second mayor point is related to putting my body in another part of the globe. Although Madrid is a nice place for meeting people from other countries, learn about their culture, what do they like/dislike, the way they think, etc., the truth is I did not find myself in such as much interaction as I would have wanted. Research has allowed me to fully experience the notion of the *Global Village*, via conferences and summer schools where I interacted with people from nearly everywhere. Also, it allowed me to live in Ireland for nine months, immersing myself completely in the daily life of the country, the Cork city and the Cork Constraint Computation Centre (4C) I was working for. I want to say that this has been one of the best experiences in my life, and it allows me to link with the final idea I am trying to express. When you know you like *A*, you do it, you enjoy it and you are happy. Good. But, there are other things *B*, *C*, *D*, etc., that you would also like, and you do not even know them. Moreover, you will probably not be available to discover them by yourself. It was research which placed me in a foreign group and country, making me discovering that this was an experience really making me happy. Thus, thanks for that.

By now I have focused in *the things* I want to say thanks for. However, it was not *research*, but concrete people, the ones making these things possible.

I would like to start acknowledging Paco. We can say that my post-graduate *story* started with the subject *Computability and Complexity*, where he was my professor. I found amazing the material he proposed, but specially the way he made us think/reason about it. Further, I discovered that, besides a great professor, he is also a great researcher (head of the GPD). I want to thank him for offering me joining the group, for trusting in my capabilities and, in summary, for giving me the opportunity for doing the master and the thesis. Finally, I want to thank him for, even with such a tight agenda, always being in charge of ensuring that everything was ok with me.

I met Fernando when I joined the GPD, and he has been my advisor during all these post-graduate years. As we have had a daily relationship since then, I appreciate him so much, both from a personal and a professional point of view, so I find difficult not to mix the two things now. I want to thank him for introducing me to the knowledge area of Constraint Functional Logic Programming over Finite Domains. He has guided me during the entire research process, at any point, at any moment. He has shown me how to carry out a project, and how to accomplish it on time and quality by applying patience, common sense and hard work. I want to thank him for his positive attitude, his advices and for being a right person. During all these years, and specially when the things turned more difficult, I have felt his confidence on me to overcome the difficulties and achieve the proposed goals. In summary, I feel very happy of having had the opportunity of working with him.

All I said for Fernando is also applicable to Teresa, who I met as well when I joined the GPD. Although she retired a couple of years ago, she has been crucial in the development of my master and in the early stages of this thesis. I want to thank her for teaching me the basics of researching, guiding me to bridge the gap between under-graduate and post-graduate. Looking back now, I am totally sure that, without her advises and support, I would have not achieved it.

Enrique, Adrián, Manu, Juan, Carlos, Roberto, Pablo, Miky, Edison, Kike, Diego, J. Fran, Javier, Alaeddin, Daniele, Tama and Vicenzo have been my colleagues at the lab office. I want to thank them for the great atmosphere we have had all this time, for being such great guys and for turning hard days into better ones.

I met Barry in the CP Summer School of 2008, where he was the professor of some lectures. I also met there PhD students either doing their thesis at the 4C, or going there as visitors. All of them told me good things about the group,

the research topics they were working at and the way they were treated. Thus, as the PhD scholarship I had at the GPD allowed me for moving abroad as a visitor, I applied to the 4C. I have visited them in 2011 and 2013. In both cases, Barry has been my supervisor. I want to thank him for allowing me to join one of the research lines of the group. For me, it has been amazing to be advised by him, to jump into a new research topic and to be able to make some progress, as to contribute something. Besides that, I hereby confirm all the good feedback I received from the 4C, and I would encourage other PhD students to visit the group, as it was such a great experience to me. In particular, besides Barry, I also want to thank to all my colleagues at the lab office, for treating me incredibly well and make me feel at home. Thanks to Diarmuid, Milan, Alex, Barry, Luis, Yuri, Yulia, Vincent, Massimo, Georgiana, Óscar, Giovanni, Lars, Mohamed, Deepak and Padraig.

Finally, I would like to acknowledge this thesis to all my family and friends, for making me a happy person. In particular, I would like to thank my parents, Pilar and Antonio, for all their love and support, and my sister, Alicia, for the same reasons, and also for helping me with the start of this section :-)

Contents

List of Figures

Abstract: The constraint satisfaction and optimization problems (CSP's and COP's, respectively) have been extensively studied in the last decades because of their importance in industry. They might imply either a complex formulation or a high computational effort, or even both. The knowledge area of Constraint Programming over Finite Domains (CP($\mathcal{FD}$)) has been identified as successful for modeling and solving these problems, as it captures their constraint-oriented nature in a succinct way. Within CP($\mathcal{FD}$), the four different paradigms Algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$), Constraint Logic Programming (CLP($\mathcal{FD}$)) and Constraint Functional-Logic Programming (CFLP($\mathcal{FD}$)) rely on a constraint solver, but they differ in the modeling language being used. In particular, CFLP($\mathcal{FD}$) provides highly expressive languages, supporting and even increasing the modeling capabilities of the declarative logic and functional languages. However, although CFLP($\mathcal{FD}$) is thus a suitable paradigm for tackling CSP's and COP's, the literature lacks case studies if compared to Algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$).

The main goal of this thesis is to encourage the use of CFLP($\mathcal{FD}$) for tackling real-life CSP's and COP's. To do so, the research is divided into three parts. First, the performance improvement of CFLP($\mathcal{FD}$). Second, showing real-life applications of CFLP($\mathcal{FD}$). And, third, performing an in-depth modeling and solving comparison of CFLP($\mathcal{FD}$) w.r.t. Algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$). The state-of-the-art CFLP($\mathcal{FD}$) system $\mathcal{TOY}(\mathcal{FD})$ is selected, which is implemented in SICStus Prolog and supports $\mathcal{FD}$ constraint solving.

The first part of the research successfully improves the solving performance of $\mathcal{TOY}(\mathcal{FD})$ by developing a generic scheme for interfacing external C++ CP($\mathcal{FD}$) solvers. Two new versions of the system are implemented by instantiating the scheme with Gecode and ILOG Solver. Also, the $\mathcal{TOY}(\mathcal{FD})$ language is enhanced with new search primitives, allowing a better specification of *ad hoc* search strategies and exploiting the structure of a problem, thus requiring less search exploration to find solutions.

Once improved the solving performance of $\mathcal{TOY}(\mathcal{FD})$, the second part of the research presents two successful real-life applications of $\mathcal{TOY}(\mathcal{FD})$. First, an Employee Timetabling Problem (ETP), coming from the communication industry, and which is modeled and solved with $\mathcal{TOY}(\mathcal{FD})$. And, second, an empirical analysis of the hardness of the classical Bin Packing Problem (BPP), for which both heuristics and CP($\mathcal{FD}$) techniques are used (the latter including $\mathcal{TOY}(\mathcal{FD})$) to solve a generated benchmark suite. This problem is quite well suited to generalized BPP instances coming from the data centre optimization industry.

Finally, the third part of the research uses the real-life problem to perform an in-depth modeling and solving comparison among the state-of-the-art algebraic CP($\mathcal{FD}$) systems Minizinc and ILOG OPL, the C++ CP($\mathcal{FD}$) systems Gecode and ILOG Solver, the CLP($\mathcal{FD}$) systems SICStus Prolog and SWI-Prolog, and the CFLP($\mathcal{FD}$) systems PAKCS and $\mathcal{TOY}(\mathcal{FD})$, showing that $\mathcal{TOY}(\mathcal{FD})$ is competitive w.r.t. any of the them.

Keywords: Constraint Programming over Finite Domains, Constraint Functional Logic Programming, Constraint Solver Integration, Search Strategies, Real-Life Employee Timetabling Problem, Bin Packing Problem, Parameterized Benchmark Generator, Algebraic Constraint Programming, Object Oriented Constraint Programming, Constraint Logic Programming.

Part I

Introduction and Preliminaries

This part of the thesis motivates and introduces the research being accomplished. Chapter 1 motivates the research, describing the contributions achieved in the three parts it is divided. Chapter 2 presents some preliminary concepts for each of these research parts.

Chapter 1

Introduction

The technological evolution, influenced by economic necessities, have turned logistics into a key issue for the success of any company or organization. Behind this abstract idea there are concrete problems, whose combinatorial nature often makes them *NP-complete* (i.e., with no general polynomial time solution methods applicable to them) [76]. Thus, time and expertise are required, both in the problem specification and in the design of the algorithm solving it.

Constraint Satisfaction and Optimization Problems (CSP's and COP's, respectively) [192] provide an abstract formalization for the problems related to the allocation and scheduling of resources, and they are present in manufacturing and service industries, such as procurement, production, transportation, distribution, information processing and communication [149]. A CSP is defined by the tuple (V, D, C), where V is the variable set $\{v_1, \ldots, v_n\}$, D is the set of n domains $\{d_1, \ldots, d_n\}$ (each d_i representing the possible values that v_i can take), and C is the constraint set (each of them placed over a subset of V, and intensionally stating the feasible combination of values these variables can take). A solution of the problem is an assignment of variables V to values of D, in such a way that the constraints of C are entailed. Similar to CSP's, a COP is defined by the tuple (V, D, C, F), where the additional parameter F represents a cost function (i.e., an expression to be minimized/maximized). A solution of a COP is an assignment of variables V to values of D, in such a way that the constraints of C are entailed and the cost function F is minimized/maximized.

CSP's and COP's have been extensively studied in the last decades, and different approaches have been applied to tackle them: The knowledge area of *Mathematical Programming* (MP) have applied both *Linear Programming* (LP) [179] and *(Mixed) Integer Programming* (MIP) [113] techniques. When applying an exhaustive search become impractical (due to a huge search space), the knowledge area of *Heuristics* have applied incomplete search-based approaches, as *ad hoc strategies* [146] or even *hyper-heuristics* [39] (providing heuristics to determine the final heuristics to apply to the problem).

The literature contains multiple applications of these knowledge areas to real-life CSP's and COP's. For example, just focusing on the last three years and on scheduling problems, some MP publications include [102], [185] and [55], and some Heuristics publications include [34], [37] and [165].

Besides MP and Heuristics, the area of knowledge of *Constraint Programming over Finite Domains* (CP($\mathcal{FD}$)) [63], [163] has been identified as specially successful for modeling and solving CSP's and COP's, as it captures their constraint-oriented nature in a succinct way. CP($\mathcal{FD}$) distinguishes between the modeling language being used to specify the problem and the techniques applied to solve it.

Any CP($\mathcal{FD}$) system is based on the notion of an $\mathcal{FD}$ *constraint solver*. It results from the combination of a *constraint store*, home for the (V, D, C) or (V, D, C, F) of the CSP or COP being specified, and a *constraint engine*, applying both *constraint propagation* and *search exploration* techniques to find feasible or optimal solutions to the problem. Briefly, the propagation of a constraint c, involving the variables $[v_{i1}, \ldots, v_{ik}]$, is an inference process removing these domain values not satisfying the constraint. As the reasoning mechanism considers each constraint of the network C separately, not all values that remain in the domains necessarily are part of some solution. Thus, the solving process is enhanced with a search exploration, modifying the initial CSP or COP by adding new constraints and reason again on it. Systematic search explorations are referred to as *backtracking search algorithms*, and they can be seen as traversing a search tree, where each node contains a modified version of the original CSP or COP. A tree node represents either a solved CSP (in which case the search stops, and the solution is reported), a failed one (in which case, the search process backtracks in the tree), or a stable CSP still containing some unbound variables (in which case, an unbound variable is selected, and the children of the node are generated and explored in a mutually exclusive and exhaustive way).

The way in which variables, domains, constraints and cost functions are specified depends on the modeling language being used. Depending on the context the problem is modeled for, there are different weights for collateral issues, such as expressivity, scalability, maintenance, integration into larger applications, etc. Most of these tasks are mutually exclusive, and thus each single modeling language provides a tradeoff among them. To present the set of paradigms identified as successful for tackling CSP's and COP's, a first distinction is done between *declarative languages* and *imperative languages*. A declarative program describes the properties a solution of the problem must hold, whereas an imperative program describes a sequence of steps to be performed in order to build-up a solution of the problem.

The integration of CP($\mathcal{FD}$) into the imperative *object oriented* paradigm [33], more specifically into the C++ language, has set up the *C++ CP($\mathcal{FD}$)* paradigm. Problem specification takes advantage of modeling features such as abstraction, encapsulation, inheritance and polymorphism. Also, the high efficiency of C++ [36] allows the CP($\mathcal{FD}$) libraries implemented in this language to obtain a higher solving performance. In any

case, it must be pointed out that a CP(FD) model is declarative, even when built via an imperative language. Two state-of-the-art C++ CP($\mathcal{FD}$) systems are Gecode [78] and IBM ILOG Solver [12].

Within declarative programming, a distinction is done between general purpose languages (being thus Turing complete [101]) and the specific purpose ones (or Turing incomplete). The integration of CP($\mathcal{FD}$) into specific purpose languages based on algebraic formulations has set up the *algebraic CP($\mathcal{FD}$)* paradigm. Problem specification takes advantage of modeling features such as the combination of basic constraints to obtain complex ones, new constraint definitions via predicates, enumerated types, array and set data structures, and an isolation from the general model to the instance-dependent input data. Also, the CSP or COP specification is abstracted from the concrete solver being used, thus allowing to try the same model over different solvers with no extra effort. Two state-of-the-art algebraic CP($\mathcal{FD}$) systems are MiniZinc [142] and IBM ILOG OPL [193].

Within general purpose declarative languages, the integration of CP($\mathcal{FD}$) into Logic Programming [120] has set up the *Constraint Logic Programming: CLP($\mathcal{FD}$)* paradigm [111]. Problem specification takes advantage of a high expressivity, including logic features such as relational notation, non-determinism, backtracking, logical variables, domain variables, and the capability of *reasoning with models* (to dynamically retract and repost the CSP or COP to the constraint store). Two state-of-the-art CLP($\mathcal{FD}$) systems are SICStus Prolog [178] and SWI-Prolog [190], each of them integrating CP($\mathcal{FD}$) via a host library `clpfd`.

Finally, the integration of CP($\mathcal{FD}$) into the multi-paradigm Functional Logic Programming (FLP) [161], [91], [19], resulting from the integration of LP and Functional Programming (FP) [128], has set up the *Constraint Functional-Logic Programming: CFLP($\mathcal{FD}$)* paradigm. In terms of modeling, the languages provided by CFLP($\mathcal{FD}$) represent probably the most complete approach within CP($\mathcal{FD}$) systems. First, their declarative nature abstracts the sequence of steps to be performed in order to build-up a solution of the problem, representing an advantage w.r.t. imperative C++ CP($\mathcal{FD}$) systems. Second, their general purpose nature allows the integration of the model into larger applications, in contrast to algebraic CP($\mathcal{FD}$) systems. Third, their high expressiveness increases the one of CLP($\mathcal{FD}$) systems, by adding FP features such as functional notation, curried expressions, higher-order functions, patterns, partial applications, lazy evaluation, types, polymorphism and constraint composition. Two state-of-the-art algebraic CFLP($\mathcal{FD}$) systems are PAKCS [93] (one of the available system versions of the Curry [92] language) and $\mathcal{TOY}(\mathcal{FD})$ [72], [124], [40].

With such huge variety of paradigms available, nowadays there is a big and alive CP($\mathcal{FD}$) community, building up a large number of systems and applications. Several conferences and journals include CP($\mathcal{FD}$) among their topics, with the following as some representative examples: *Constraints*, International Conference on Principles and Practice of Constraint Programming: *CP*, International Conference on Integration

of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming: *CPAIOR*, Theory and Practice of Logic Programming: *TPLP*, International Joint Conferences on Artificial Intelligence: *IJCAI*, European Conference on Artificial Intelligence: *ECAI*.

These CP($\mathcal{FD}$) related conferences and journals present multiple real-life applications. However, whereas they are more or less split among algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$) approaches, it seems that CFLP($\mathcal{FD}$) has not attracted the attention of the CP($\mathcal{FD}$) community. For example, by selecting the last two years editions of these conferences and journals, no single CFLP($\mathcal{FD}$) application is found. With respect to the algebraic CP($\mathcal{FD}$) system MiniZinc, applications include:

- *MiningZinc* [88]: A general framework for constraint-based pattern mining (one of the most popular tasks in data mining). It provides a novel library of functions and constraints to support modeling pattern mining tasks in the MiniZinc language.
- *Subproblem Dominance* [56]: A catching technique for detecting and storing nogoods to improve the search exploration. The experiments rely on a set of MiniZinc models, which are catched and executed in CHUFFED [57].
- *Embarrassingly Parallel Search* [156]: A method for solving CP problems in parallel. It uses a benchmark of non-trivial problems modeled in MiniZinc to show the goodness of the approach.
- *MiniZinc Globalizer* [118]: A method that, given a constraint model, suggests global constraints to replace parts of it (helping non-expert users to write higher-level models). It is implemented for the MiniZinc language.
- *MiniZinc with Functions* [189]: An extension of the MiniZinc language to support functions (allowing to develop more elegant and readable models).
- *Optimization for Software Developers* [73]: An alternative interface to CP technology based in Java, hiding from the user all reference to CP specific concepts. It is based in a compilation from this Java model to a MiniZinc one.

Similarly, w.r.t. the C++ CP($\mathcal{FD}$) system Gecode, applications include:

- *Balancing Bike Sharing* [77]: A CP and a Large Neighborhood Search algorithms implemented in Gecode for tacking a Balancing Bike Sharing System.
- *Laser Cutting Path Planning* [117]: A Gecode approach for tacking a problem coming from the sheet metal industry.
- *Atom Mapping* [129]: A Gecode approach for tacking a problem coming from the chemical industry.
- *Multidimensional Binpacking* [87]: A Gecode decomposition for the Multidimensional Binpacking Constraint, based on a `bin_packing` constraint for each dimension plus a set of `all_different` constraints automatically inferred.

- *CFG* [97]: A Gecode improved propagator for the Context-Free Grammar constraint.
- *Bandit Search for Constraint Programming* [125]: A Gecode adaptation of the Monte Carlo Tree-Search to the specifics of CP search trees.

Finally, w.r.t. the CLP($\mathcal{FD}$) system SICStus `clpfd`, applications include:

- *FDCC* [22]: A SICStus `clpfd` approach for solving constraint systems involving arrays (with accesses, updates and size constraints) and $\mathcal{FD}$ constraints over their elements and indexes.
- *The Painting Fool* [59]: An automated painter with several features, using SICStus `clpfd` for scene construction (by considering the requests of the user as a CSP).
- *Matrix CP Models* [24]: A SICStus `clpfd` approach including two methods for improving propagation in matrix CP models (evaluated under Nurse Rostering benchmarks).
- *A Scalable Sweep Algorithm for the Cumulative Constraint* [119]: A SICStus `clpfd` and Choco filtering algorithm for the `cumulative` global constraint.
- *Improving all_different + sum* [25]: A SICStus `clpfd` filtering algorithm for the combination of an `all_different` global constraint and an inequality between a sum of variables and a constant.
- *25 Years of SICStus Prolog* [44]: An overview of the first 25 years of SICStus Prolog, enumerating some of their real-life applications, which includes the use of `clpfd` for the optimization engine of the logistics RedPrairie Corporation.

Given this lack of real-life applications for CFLP($\mathcal{FD}$), this thesis is focused on an empirical evaluation of the applicability of CFLP($\mathcal{FD}$) to tackle real-life CSP's and COP's. A CFLP($\mathcal{FD}$) modeling language and the overhead of its solving performance are analyzed in detail. These modeling and solving results are also compared to those of other state-of-the-art algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$) systems.

The research being accomplished is focused on the CFLP($\mathcal{FD}$) system $\mathcal{TOY}(\mathcal{FD})$, implemented in SICStus Prolog, and solving syntactic equalities and disequalities (via a Herbrand solver: $\mathcal{H}$), as well as $\mathcal{FD}$ constraints (via a CP($\mathcal{FD}$) solver). The first part of the research focuses on the performance improvement of $\mathcal{TOY}(\mathcal{FD})$ (maintaining and even increasing the expressiveness of its language) by interfacing the C++ CP($\mathcal{FD}$) solvers of Gecode and ILOG Solver, and specifying *ad hoc* search strategies. Then, relying on the higher performance achieved, the second part of the research focuses on describing two real-life applications of $\mathcal{TOY}(\mathcal{FD})$, an Employee Timetabling Problem (coming from the communication industry) and an empirical analysis of the hardness of the classical Bin Packing Problem (to further solve generalized instances coming from the data centre industry). Finally, the third part of the research focuses on positioning $\mathcal{TOY}(\mathcal{FD})$ w.r.t. the algebraic CP($\mathcal{FD}$) systems MiniZinc and ILOG OPL, the C++

CP($\mathcal{FD}$) systems Gecode and ILOG Solver and the CLP($\mathcal{FD}$) systems SICStus Prolog and SWI-Prolog (also including in the analysis the other state-of-the-art CFLP($\mathcal{FD}$) system, PAKCS).

Next, Sections 1.1 and 1.2 present the contributions and publications achieved on each of the three research parts. Then, Section 1.3 presents the structure of the thesis, giving an overview of the content of each chapter.

1.1 Contributions

The first part of the research seeks the improvement of the solving performance of $\mathcal{TOY}(\mathcal{FD})$. Initially, the system uses the host solver SICStus `clpfd` (denoting the system version as $\mathcal{TOY}(\mathcal{FD}s)$). In this thesis, the state-of-the-art C++ CP($\mathcal{FD}$) solvers Gecode and ILOG Solver are interfaced to $\mathcal{TOY}(\mathcal{FD})$, developing the new system versions $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$, respectively. For them, the $\mathcal{TOY}(\mathcal{FD})$ language is enhanced to allow the specification of *ad hoc* search strategies. The contributions are the following:

- Develop a scheme for interfacing C++ CP($\mathcal{FD}$) solvers into $\mathcal{TOY}(\mathcal{FD})$, in a setting applicable to other CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems implemented in Prolog. The scheme is shown to be generic enough, interfacing Gecode and ILOG Solver by finding no extra interface difficulties but the ones described in the scheme.
  - The different commands $\mathcal{TOY}(\mathcal{FD})$ requests to the CP($\mathcal{FD}$) solver (to coordinate it) are identified, creating an abstract and extensible interface between system and solver (which includes gluing the Prolog and C++ components). The management of the mismatch between the system and solver different variable, constraint and type representations is described. The C++ CP($\mathcal{FD}$) solver is adapted to fulfill the CFLP($\mathcal{FD}$) requirements of model reasoning, multiple search strategies (interleaved with constraint posting) and both incremental and batch propagation modes.
- Enhance the language of the new developed $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ versions with eight new parameterized search primitives (as they are the ones providing better solving performance), providing a more detailed search specification to the solver (in a setting applicable to other CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems implemented in Prolog and interfacing external C++ CP($\mathcal{FD}$) solvers).
  - The primitives include novel search concepts (allowing incomplete search) not directly available in the primitives provided by Gecode and ILOG Solver (obviously, these libraries can be extended, so that new search primitives can be implemented on them): Performing an exhaustive breadth exploration of the search tree (further sorting the satisfiable solutions by a specified criterion). Fragmenting the variables by pruning each one to a subset

of its domain values (instead of binding it to a single value). Applying the labeling or fragment strategy only to a subset of the variables involved.

Also, some of the search criterion of the primitives can be directly specified at $\mathcal{TOY}(\mathcal{FD})$ level. Moreover, the use of $\mathcal{TOY}(\mathcal{FD})$ allows to easily combine several primitives (to built up complex search strategies), as well as use model reasoning to apply different search scenarios to the solving of a problem.

- Measure the performance of the different $\mathcal{TOY}(\mathcal{FD})$ versions, using a minimal benchmark based on three classical CSP's (Magic Series, Queens and Langford's Number) and a classical COP (Golomb Rulers). The results reveal that $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ clearly outperform $\mathcal{TOY}(\mathcal{FD}s)$ in terms of solving performance. Moreover, the new search primitives allow to improve the performance of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$.
  - The set of problems is complete enough, as it covers the whole repertoire of $\mathcal{FD}$ constraints supported by $\mathcal{TOY}(\mathcal{FD})$. Also, by using different instances per problem the experiments analyze the $\mathcal{TOY}(\mathcal{FD})$ performance as the hardness of the problem scales. Finally, the analysis of the search primitives to be applied for each problem is based on the structure of their solutions found before.

The second part of the research presents two real-life applications of $\mathcal{TOY}(\mathcal{FD})$, an Employee Timetabling Problem (ETP) and an empirical analysis of the hardness of the classical Bin Packing Problem (BPP). The contributions are the following:

- Describe a non-monolithic algorithm for specifying a generic version of the ETP, where the workers (which denote the employees) are split into different teams. Compare its solving performance results with the ones of the classical benchmarks used before.
  - The complex formulation exploits the high expressivity of $\mathcal{TOY}(\mathcal{FD})$. This formulation is fully parametric in different aspects of the problem, as the number of days of the timetable, number of teams (and number of workers per team), periodicity the extra worker can be selected (and the extra factor its working hours must be paid), number of different kinds of working days (and the concrete shifts requested on each of them), absences of the regular workers of the teams and the tight the shifts must be distributed among the workers of a team.
  - The solving approach presented is based in problem decomposition, splitting the search space into as many possible assignments of teams to days, exploring only those ones that are feasible and, for each of them, splitting again its search space subset into as many independent problems (exponentially easier to be solved) as teams there are.

  - The performance results (both in terms of the concrete $\mathcal{TOY}(\mathcal{FD})$ version and of the application of an *ad hoc* strategy) are similar to the ones for the classical benchmark problems, but the differences between the instances are greater.

- Solve a parametric generated BPP benchmark (based on the well known Weibull distribution [201]), applying two equivalent CP($\mathcal{FD}$) models (Gecode and $\mathcal{TOY}(\mathcal{FD}g)$) and four heuristics.

  - Weibull generates a great variety of item size distributions, as its flexibility allows to represent nearly any unimodal distribution. Its parametric model is used to very accurately fit real-life BPP instances. Maximum Likelihood Fitting and Quantile-Quantile plots are used to observe the quality of the fit. The Kolmogorov-Smirnov and χ^2 statistical tests rigorously ensure it.
  - The benchmark is built up using 199 combinations of the Weibull parameters (generating 100 instances per combination). Also, eleven different scenarios are proposed, setting the size of the bin to the size of the highest item of the instance times a factor ranging from 1.0 to 2.0 (increasing it 0.1 on each new scenario). Finally, scripting techniques are used to set up benchmark solving sessions for the CP($\mathcal{FD}$) systems and the heuristics.
  - The analysis results reveal that both CP($\mathcal{FD}$) and the heuristics are suitable to solve the problem, as depending on the concrete instance (Weibull parameters) and bin size (scenario) chosen, both techniques provide a different tradeoff between the elapsed time for solving the instance and the quality of the solution achieved.

The third part of the research positions $\mathcal{TOY}(\mathcal{FD})$ w.r.t. the state-of-the-art algebraic CP($\mathcal{FD}$) systems MiniZinc and ILOG OPL, the C++ CP($\mathcal{FD}$) systems Gecode and ILOG Solver, the CLP($\mathcal{FD}$) systems SICStus Prolog and SWI-Prolog, and the CFLP($\mathcal{FD}$) system PAKCS. It encourages the use of $\mathcal{TOY}(\mathcal{FD})$ (and the use of the paradigm CFLP($\mathcal{FD}$) itself), showing that it is competitive w.r.t. any of the other systems for the modeling and solving of two COP's, the classical Golomb and the real-life ETP one. The contributions are the following:

- Perform an in-depth modeling comparison of the two COP's among MiniZinc, ILOG OPL, Gecode, ILOG Solver, SICStus Prolog, SWI-Prolog, PAKCS and $\mathcal{TOY}(\mathcal{FD})$.

  - The Golomb benchmark, with a simple formulation, provides general insights about the abstraction of the constraint solver, the specification of the $\mathcal{FD}$ variables, $\mathcal{FD}$ constraints and search strategies, as well as the output of the solutions. The ETP, with a complex formulation (fully parametric, non-monolithic and including CP($\mathcal{FD}$) independent components), exploits the

expressive power of the different paradigms, allowing to analyze in detail the strengths and drawbacks of each of them.

  - The comparison includes several code examples, to put in context the ideas being presented. Besides that, the whole code of each model (per COP and system) is provided. Thus, the expressiveness comparison also includes the amount of lines of code being used.

- Perform an in-depth solving comparison of the two COP's among MiniZinc, ILOG OPL, Gecode, ILOG Solver, SICStus Prolog, SWI-Prolog, PAKCS, $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}s)$.

  - Set a common framework for running the experiments, considering the system versions, the global constraints being used (and their filtering algorithms) and the measurement of the elapsed time. Three instances per problem (solved in tenths of second, seconds and minutes, respectively) are considered.
  - Compare the solving performance of the ten systems, analyzing their ranking and slow-downs, and discussing the performance order existing among the systems using the same constraint solving library. Specialize the solving comparison for the Gecode, ILOG Solver and SICStus related systems, devoting an isolated analysis for each of them.
  - Provide a head-to-head comparison between $\mathcal{TOY}(\mathcal{FD}g)$ and the native Gecode model (respectively $\mathcal{TOY}(\mathcal{FD}i)$ and the native ILOG Solver model, and $\mathcal{TOY}(\mathcal{FD}s)$ and the native SICStus model), to analyze the overhead of each $\mathcal{TOY}(\mathcal{FD})$ version, justifying it.

1.2 Publications

The first part of the research has lead to the following publications:

1. **Improving the Performance of FD Constraint Solving in a CFLP System** [48].
   It has been presented in the 11th International Symposium on Functional and Logic Programming: FLOPS'12. It has been published by Springer, LNCS 7294, pages 88-103.
2. **Integrating ILOG CP Technology into TOY** [46].
   It has been presented in the 18th International Workshop on Functional and (Consraint) Logic Programming: WFLP'09. It has been published by Springer, LNCS 5979, pages 27-43.
3. **Improving the Search Capabilities of a CFLP(FD) System** [51].
   It has been presented in the XIII Spanish Conference on Programming and Computer Languages: PROLE'13, and it has been invited for its submission to the

Electronic Communications of the EASST. It has been published in the online proceedings of the conference, pages 273-287.

The second part of the research has lead to the following publications:

4. **A CFLP Approach for Modeling and Solving a Real Life Employee Timetabling Problem** [47].
   It has been presented in the 6th International Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems: COPLAS'11. It has been published in the online proceedings of the conference, pages 63-70.
5. **Weibull-Based Benchmarks for Bin Packing** [45].
   It has been presented in the 18th International Conference on Principles and Practice on Constraint Programming: CP'12. It has been published by Springer, LNCS 7514, pages 207-222.

The third part of the research has lead to the following publications:

6. **Applying CP(FD), CLP(FD) and CFLP(FD) to a Real-Life Employee Timetabling Problem** [49].
   It has been presented in the 13th International Conference on Computational Science: ICCS'13. It has been published by Procedia Computer Science, 18(0), pages 531-540.
7. **Comparing TOY(FD) with State-of-the-Art Constraint Programming Systems** [50]. It has been published as the technical report DSIC-14.13 to the Departamento de Sistemas Informáticos y Computación (DSIC) of the Complutense University of Madrid (UCM), pages 1-135.

1.3 Structure

This thesis performs an empirical evaluation of the applicability of $\mathcal{TOY}(\mathcal{FD})$ to tackle real-life CSP's and COP's, analyzing in detail the expressiveness of its modeling language and the overhead of its solving performance, and comparing both of them to the modeling and solving results of other state-of-the-art algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$) systems. The document is organized as follows:

Part I motivates the proposed research, and presents some preliminary concepts.

Chapter 1 presents the three parts of the research being accomplished. Section 1.1 describes the contributions for each of them, and Section 1.2 enumerates the publications they lead to. Section 1.3 presents the organization of the thesis, which is being described right now.

Chapter 2 presents some preliminary concepts. Section 2.1 presents a CSP and COP example. Section 2.2 presents an overview of the constraint propagation and search exploration solving techniques used by any CP($\mathcal{FD}$) constraint solver. Section 2.3 presents

the different modeling paradigms available in CP($\mathcal{FD}$), providing a description of each of the state-of-the-art systems to be further considered. Section 2.4 presents an introduction to $\mathcal{TOY}(\mathcal{FD})$, including some program examples and goals for an overview of the modeling features of the language and its operational semantics.

Part II presents the first part of the research being accomplished, which focuses on the performance improvement of $\mathcal{TOY}(\mathcal{FD})$.

Chapter 3 describes the integration of external C++ CP($\mathcal{FD}$) solvers, with state-of-the-art performance, into $\mathcal{TOY}(\mathcal{FD})$. Section 3.1 describes the architecture of $\mathcal{TOY}(\mathcal{FD})$, focusing on the interface to its $\mathcal{FD}$ solver and identifying the requirements to coordinate it. Section 3.2 describes the scheme for interfacing C++ CP($\mathcal{FD}$) solvers, and Section 3.3 instantiates it with Gecode and ILOG Solver. Section 3.4 analyzes the new $\mathcal{TOY}(\mathcal{FD})$ performance achieved by using a set of classical benchmark problems. Section 3.5 presents some related work, and Section 3.6 reports conclusions.

Chapter 4 describes the development of new search primitives, implemented in the new $\mathcal{TOY}(\mathcal{FD})$ versions interfacing C++ CP($\mathcal{FD}$) solvers. Section 4.1 presents an abstract description of the new parameterizable $\mathcal{TOY}(\mathcal{FD})$ search primitives, pointing out some novel concepts not directly available neither in Gecode nor in ILOG Solver. Also, it points out how to specify some search criterion at $\mathcal{TOY}(\mathcal{FD})$ level and how easily the strategies can be combined to set different search scenarios. Section 4.2 describes the implementation of the primitives, presenting first an abstract view of the $\mathcal{TOY}(\mathcal{FD})$ requirements, and how they are targeted to the Gecode and ILOG Solver libraries. It also evaluates the impact of the search strategies implementation in the architecture of the system. Section 4.3 analyzes the new $\mathcal{TOY}(\mathcal{FD})$ performance achieved, revisiting the classical benchmark problems used before. For each CSP and COP, it shows that the use of the search strategies improve the solving performance. Section 4.4 presents some related work, and Section 4.5 reports conclusions.

Part III presents the second part of the research being accomplished, which focuses on two real-life applications of $\mathcal{TOY}(\mathcal{FD})$.

Chapter 5 describes the modeling and solving of an Employee Timetabling Problem. Section 5.1 presents a description of the problem and Section 5.2 describes the solving approach to tackle it. Section 5.3 presents a parametric algorithm implementing the solving approach, which is the one being followed by $\mathcal{TOY}(\mathcal{FD})$ to model the problem. Section 5.4 presents the results of running different instances of the problem in the different $\mathcal{TOY}(\mathcal{FD})$ versions, and compares these results with the ones obtained for solving the classical benchmark problems. Section 5.5 presents some related work, and Section 5.6 reports conclusions.

Chapter 6 describes the application of both CP($\mathcal{FD}$) and heuristics techniques to an empirical analysis process to test the hardness of the Bin Packing Problem. Section 6.1 discusses the parametrical statistical model provided by the Weibull distribution, showing the variety of item size distributions that can be generated with it. It proves the model to be successful on fitting real-life generalized instances coming from the

data centre optimization and education industries. Section 6.2 presents the empirical analysis process layout. It describes the set of instances created (which gives support for very controlled experiments) and both the heuristics and CP($\mathcal{FD}$) models used to run the instance set. Section 6.3 analyzes the results by the CP($\mathcal{FD}$) and heuristics methods, focusing on the relation between the Weibull parameters of the instances and the quality of the solutions. Section 6.4 presents some related work, and Section 6.5 reports conclusions.

Part IV presents the third part of the research being accomplished, which focuses on positioning $\mathcal{TOY}(\mathcal{FD})$ w.r.t. state-of-the-art algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$) systems for the modeling and solving of two COP's, the classical Golomb benchmark and the real-life timetabling previously described.

Chapter 7 describes an in-depth modeling comparison of the two COP's between the algebraic CP($\mathcal{FD}$) systems MiniZinc and ILOG OPL, the C++ CP($\mathcal{FD}$) systems Gecode and ILOG Solver, the CLP($\mathcal{FD}$) systems SICStus Prolog and SWI-Prolog, and the CFLP($\mathcal{FD}$) systems PAKCS and $\mathcal{TOY}(\mathcal{FD})$. Section 7.1 uses the Golomb problem to provide general insights about the abstraction of the constraint solver, the specification of the $\mathcal{FD}$ variables, $\mathcal{FD}$ constraints and search strategies, as well as the output of the solutions. Then, next sections use the real-life timetabling problem to discuss in detail how each paradigm tackles a concrete modeling issue: Section 7.2 analyzes the coordination of the different stages. Section 7.3 focuses on the data structures. Section 7.4 on the variables. Section 7.5 on the constraints, and Section 7.6 provides a final expressiveness comparison. To end the chapter, Section 7.7 examines the amount of lines of code needed by each system for both COP's. Section 7.8 presents some related work, and Section 7.9 reports conclusions.

Chapter 8 performs an in-depth solving comparison of the same problems and systems (in the case of $\mathcal{TOY}(\mathcal{FD})$, considering the different system versions). Section 8.1 sets the context for the solving comparison. Section 8.2 presents the performance of all the systems, analyzing the ranking and slow-down results. It also discusses the performance ranking among the Gecode, ILOG Solver, SICStus `clpfd` and SWI-Prolog `clpfd` constraint solvers, gathering the related systems into different sets. Section 8.3 presents a dedicated ranking and slow-down analysis of the Gecode related systems MiniZinc, Gecode and $\mathcal{TOY}(\mathcal{FD}g)$. It discusses the search statistics of the different systems, and provide a low-level monitoring of the search exploration performed by the $\mathcal{TOY}(\mathcal{FD})$ version and the Gecode native one, discussing their differences. Sections 8.4 and 8.5 are similar to Section 8.3, but they focus on the ILOG Solver and SICStus `clpfd` related systems, respectively. Section 8.6 presents some related work, and Section 8.7 reports conclusions.

Part V summarizes the accomplished research.

Chapter 9 presents the main conclusions achieved, with Sections 9.1, 9.2 and 9.3 devoted to the first, second and third parts of the research, respectively. Finally, Section 9.4 presents some perspectives for them.

Chapter 2

Preliminaries

This chapter presents some preliminary concepts for the three parts of the research being accomplished. It is organized as follows: Section 2.1 presents a brief introduction to CSP's and COP's. Section 2.2 presents an overview of the constraint propagation and search exploration solving techniques used by any CP($\mathcal{FD}$) constraint solver. Section 2.3 presents the different modeling paradigms available in CP($\mathcal{FD}$), providing a description of each of the state-of-the-art systems to be further considered. Finally, Section 2.4 presents an introduction to $\mathcal{TOY}(\mathcal{FD})$, including some program and goal examples to illustrate its language and its operational semantics.

2.1 Constraint Satisfaction and Optimization Problems

Nowadays, Constraint Satisfaction and Optimization problems (CSP's and COP's, respectively) [192] are present in manufacturing and service industries, such as procurement, production, transportation, distribution, information processing and communication. This section presents a brief introduction to these problems.

2.1.1 Constraint Satisfaction Problems

A CSP is defined by the tuple (V, D, C), where V is the variable set $\{v_1, \ldots, v_n\}$, D is the set of n domains $\{d_1, \ldots, d_n\}$ (each d_i being the finite set of possible values that v_i can take), and C is the constraint set (each of them placed over a subset of V, and intensionally stating the feasible combination of values these variables can take). A solution of the problem is an assignment of variables V to values of D, in such a way that the constraints of C are entailed. The search space S (set of candidates to be a solution) is represented by each possible value combination of the variables.

A paradigmatic CSP example is the N Queens problem, consisting of a puzzle placing N chess queens on an $N \times N$ chessboard, so that no two queens attack each other

(i.e., no two queens share the same row, column, or diagonal). As an example, top part of Figure 2.1 presents a CSP specification of the Queens-4 problem instance (with four queens), with the bottom part presenting the initial search space (highlighting in boldface its two feasible solutions, and providing a graphical representation for them). On it, the domain of each variable v_i is constrained to be in $a..b$ (where a and b are the lower and upper bounds of a range of integers), and the symbol != stands for a disequality constraint.

Instead of using a couple of variables v_{ki}, v_{kj} to represent the coordinates (i, j) each queen k takes, the specification uses a single variable per queen. With this, it directly matches the problem requirement of placing N queens on an $N \times N$ chessboard, with no two queens placed on the same column. The *i*-th queen is directly assigned to column i, with the variable v_i representing its row. The initial domain of each variable is set to $1..N$, as each queen can be placed at any row.

The remaining problem requirements are specified by using $3 \times (N * (N-1))/2$ constraints. A third of these constraints follow the pattern v_i != v_j, ensuring that no two queens are placed in the same row. Another third of them follow the pattern v_i != $v_{j+t} + t$, ensuring that no two queens share the same decreasing diagonal. The last third part follow the pattern v_i != $v_{j-t} - t$, ensuring that no two queens share the same increasing diagonal.

The combinatorial nature of the problem gives rise to a search space of N^N candidates (as the problem contains N variables, each one with an initial domain of N values). Even in a tiny instance as Queens-4, this leads to 256 candidates, for which only 2 are feasible solutions. As it can be seen in Figure 2.1, the two feasible solutions are, in some sense, symmetric, as they represent exactly the same positions for the different queens, just turning the chess board. Thus, a symmetry breaking constraint as, for example, $v_1 < v_4$, would have removed the symmetry, leading to a single solution (which is better than mixing up the user with two solutions that are in fact the same one).

2.1.2 Constraint Optimization Problems

A COP is defined by the tuple (V, D, C, F), where the additional parameter F represents a cost function (i.e., an expression to be minimized/maximized). A solution of a COP is an assignment of variables V to values of D, in such a way that the constraints of C are entailed and the cost function F is minimized/maximized.

A paradigmatic COP example is the N Golomb Rulers problem, which is said to have many practical applications including sensor placements for x-ray crystallography and radio astronomy [65]. It consists of a puzzle placing N marks ($0 = m_0 < m_1 < \ldots < m_{N-1}$) in a ruler, such that the distances $d_{i,j} = m_j - m_i$ for $0 \leq i < j < N$ are pairwise distinct. An optimal ruler is the one minimizing the mark m_{N-1}. As an example, top part of Figure 2.2 presents a possible COP specification of the Golomb-5 problem instance

$V \equiv \{v_1, v_2, v_3, v_4\}$;
$D \equiv \{v_1$ in $1..4, v_2$ in $1..4, v_3$ in $1..4, v_4$ in $1..4\}$;
$C \equiv \{v_1 \mathrel{!}= v_2, v_1 \mathrel{!}= v_3, v_1 \mathrel{!}= v_4, v_2 \mathrel{!}= v_3, v_2 \mathrel{!}= v_4, v_3 \mathrel{!}= v_4,$
$v_1 \mathrel{!}= v_2 + 1, v_1 \mathrel{!}= v_3 + 2, v_1 \mathrel{!}= v_4 + 3,$
$v_2 + 1 \mathrel{!}= v_3 + 2, v_2 + 1 \mathrel{!}= v_4 + 3,$
$v_3 + 2 \mathrel{!}= v_4 + 3,$
$v_1 \mathrel{!}= v_2 - 1, v_1 \mathrel{!}= v_3 - 2, v_1 \mathrel{!}= v_4 - 3,$
$v_2 - 1 \mathrel{!}= v_3 - 2, v_2 - 1 \mathrel{!}= v_4 - 3,$
$v_3 - 2 \mathrel{!}= v_4 - 3\}$;

$S \equiv \{$**(v_1 = 1, v_2 = 1, v_3 = 1, v_4 = 1), ..., (v_1 = 1, v_2 = 1, v_3 = 1, v_4 = 4),**
(v_1 = 1, v_2 = 1, v_3 = 2, v_4 = 1), ..., (v_1 = 1, v_2 = 1, v_3 = 2, v_4 = 4),
..., (v_1 = 2, v_2 = 4, v_3 = 1, v_4 = 3), ..., (v_1 = 3, v_2 = 1, v_3 = 4, v_4 = 2), ...
(v_1 = 4, v_2 = 4, v_3 = 4, v_4 = 1), ..., (v_1 = 4, v_2 = 4, v_3 = 4, v_4 = 4)$\}$

	v_1	v_2	v_3	v_4
1			X	
2	X			
3				X
4		X		

	v_1	v_2	v_3	v_4
1		X		
2				X
3	X			
4			X	

Figure 2.1: CSP Specification and Feasible Solutions for Queens-4

(with five rulers), with the bottom part presenting the initial search space (highlighting its optimal solution, and providing a graphical representation for it).

Besides using N variables to represent the ruler marks $[m_0, \ldots m_{N-1}]$, $N*(N-1)/2$ additional variables are used to represent the distances between each pair of marks $[d_{1,0}, \ldots, d_{(N-1),0}, d_{2,1}, \ldots, d_{(N-1),(N-2)}]$ (leading to a total amount of $(N^2 + N)/2$ variables). An upper bound of $2^{N-1} - 1$ for m variables is known [173]. It relies in the fact that, if the distance between m_j and m_{j+1} is always 2^{j+1}, then each m_i can be assigned to $2^i - 1$. Thus, the bit representation of any $d_{j,i}$ = m_j - m_i contains zero in the least i bits, followed by $j - i$ ones. As it can be seen, this bit representation is different for each each of the d variables, so their values are different as well [173]. By setting m variables to an initial domain of $0..(2^{N-1} - 1)$, the distance of each pair of marks must be in this domain as well, so d variables are set to an initial domain of $0..(2^{N-1} - 1)$ too. The remaining problem requirements are specified by using the following N^2 constraints: One constraint is used to assign m_0 to 0. $N - 1$ constraints state each m_i to be smaller than m_{i+1}. $N * (N - 1)/2$ constraints assign each $d_{i,j}$ to the subtraction of its two associated marks. Finally, a same amount of constraints state that d variables are pairwise different.

In addition to these N^2 constraints, the specification contains some more

– –

$V \equiv [m_0, m_1, m_2, m_3, m_4,$
$d_{1,0}, d_{2,0}, d_{3,0}, d_{4,0}, d_{2,1}, d_{3,1}, d_{4,1}, d_{3,2}, d_{4,2}, d_{4,3}]$;

$D \equiv [m_0$ in $0..15, m_1$ in $0..15, m_2$ in $0..15, m_3$ in $0..15, m_4$ in $0..15,$
$d_{1,0}$ in $0..15, d_{2,0}$ in $0..15, d_{3,0}$ in $0..15, d_{4,0}$ in $0..15, d_{2,1}$ in $0..15,$
$d_{3,1}$ in $0..15, d_{4,1}$ in $0..15, d_{3,2}$ in $0..15, d_{4,2}$ in $0..15, d_{4,3}$ in $0..15]$;

$C \equiv [m_0 = 0,$
$m_0 < m_1, m_1 < m_2, m_2 < m_3, m_3 < m_4,$
$d_{1,0} = m_1 - m_0, d_{2,0} = m_2 - m_0, d_{3,0} = m_3 - m_0, d_{4,0} = m_4 - m_0,$
$d_{2,1} = m_2 - m_1, d_{3,1} = m_3 - m_1, d_{4,1} = m_4 - m_1,$
$d_{3,2} = m_3 - m_2, d_{4,2} = m_4 - m_2$
$d_{4,3} = m_4 - m_3$
$d_{1,0} != d_{2,0}, d_{1,0} != d_{3,0}, d_{1,0} != d_{4,0}, d_{1,0} != d_{2,1}, d_{1,0} != d_{3,1}, d_{1,0} != d_{4,1}, d_{1,0} != d_{3,2}, d_{1,0} != d_{4,2},$
$d_{1,0} != d_{4,3}, d_{2,0} != d_{3,0}, d_{2,0} != d_{4,0}, d_{2,0} != d_{2,1}, d_{2,0} != d_{3,1}, d_{2,0} != d_{4,1}, d_{2,0} != d_{3,2}, d_{2,0} != d_{4,2},$
$d_{2,0} != d_{4,3}, d_{3,0} != d_{4,0}, d_{3,0} != d_{2,1}, d_{3,0} != d_{3,1}, d_{3,0} != d_{4,1}, d_{3,0} != d_{3,2}, d_{3,0} != d_{4,2}, d_{3,0} != d_{4,3},$
$d_{4,0} != d_{2,1}, d_{4,0} != d_{3,1}, d_{4,0} != d_{4,1}, d_{4,0} != d_{3,2}, d_{4,0} != d_{4,2}, d_{4,0} != d_{4,3}, d_{2,1} != d_{3,1}, d_{2,1} != d_{4,1},$
$d_{2,1} != d_{3,2}, d_{2,1} != d_{4,2}, d_{2,1} != d_{4,3}, d_{3,1} != d_{4,1}, d_{3,1} != d_{3,2}, d_{3,1} != d_{4,2}, d_{3,1} != d_{4,3}, d_{4,1} != d_{3,2},$
$d_{4,1} != d_{4,2}, d_{4,1} != d_{4,3}, d_{3,2} != d_{4,2}, d_{3,2} != d_{4,3}, d_{4,2} != d_{4,3},$
$d_{1,0} >= 1, d_{2,0} >= 3, d_{3,0} >= 6, d_{4,0} >= 10,$
$d_{2,1} >= 1, d_{3,1} >= 3, d_{4,1} >= 6,$
$d_{3,2} >= 1, d_{4,2} >= 3,$
$d_{4,3} >= 1$
$d_{1,0} < d_{4,3}]$;

$F \equiv$ minimize m_4;

– –

$S \equiv [(m_0 = 0, m_1 = 0, m_2 = 0, m_3 = 0, m_4 = 0, d_{1,0} = 0, d_{2,0} = 0, d_{3,0} = 0, \ldots d_{4,3} = 0),$
$\ldots,$ **$(m_0 = 0, m_1 = 1, m_2 = 4, m_3 = 9, m_4 = 11, d_{1,0} = 1, d_{2,0} = 4,$**
$d_{3,0} = 9, d_{4,0} = 11, d_{2,1} = 3, d_{3,1} = 8, d_{4,1} = 10, d_{3,2} = 5, d_{4,2} = 7, d_{4,3} = 2)$, $\ldots$
$(m_0 = 15, m_1 = 15, m_2 = 15, m_3 = 15, m_4 = 15, d_{1,0} = 15, \ldots d_{4,3} = 15)]$

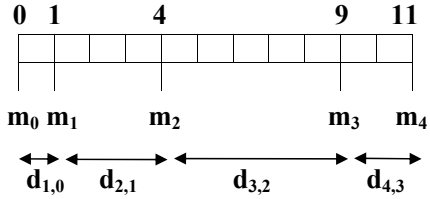

Figure 2.2: COP Specification and Optimal Solution for Golomb-5

redundant constraints, improving the performance for finding the solutions of the problem. For example, $N*(N-1)/2$ redundant constraints are used to apply a lower bound to d variables [183]: Each distance $d_{j,i}$ = m_j - m_i satisfies the property of being equal to the sum of all the distances between marks m_i and m_j (i.e., $d_{i,j}$ = (m_j - m_{j-1}) + (m_{j-1} - m_{j-2}) + ... + (m_{i+1} - m_i)). As all these distances are pairwise distinct, their sum must be, at least, the sum of the first $j - i$ integers. As an additional constraint, $d_{1,0}$ < $d_{(N-2),(N-1)}$ breaks a few symmetries in the solutions being found.

Finally, the cost function specifies that, among all the feasible solutions of the problem, only the one (or ones) containing a minimal value for m_4 should be taken into account. That is, the combinatorial nature of the problem gives rise to a search space of $(2^{N-1})^{(N^2+N)/2}$ candidates (as the problem contains $(N^2 + N)/2$ variables, each of them with an initial domain of 2^{N-1} values). Even in a tiny instance as Golomb-5, this leads to an enormous amount of $16^{15} \equiv$ 1,152,921,504,606,846,976 candidates, for which only 81 of them are feasible solutions. Regarding m_4: 38 solutions assign it to the value 15. 21 solutions to 14. 14 solutions to 13. 7 solutions to 12. Finally, only 1 solution to 11 (being thus the optimal one).

2.2 CP($\mathcal{FD}$) Solving Techniques

Constraint Programming over Finite Domains (CP($\mathcal{FD}$)) is a suitable knowledge area for tackling CSP's and COP's, as it captures their constraint-oriented nature in a succinct way. It distinguishes between the modeling language being used to specify the problem and the techniques applied to solve it. Regarding the latter, any CP($\mathcal{FD}$) system is based on the use of a constraint solver, which applies constraint propagation and search exploration to find solutions. Sections 2.2.1 and 2.2.2 present a brief introduction to these two techniques. They reuse some of the definitions and examples used in Chapters 3 and 4 of [163], which provide an extensive survey to propagation and search algorithms, respectively. More alternatives to the combination of the two techniques can be found in [23].

2.2.1 Constraint Propagation

A constraint c can be seen as a relation defined on a sequence of variables $[v_{i1}, \ldots, v_{ik}]$, whose declarative semantics is the subset of $\mathbb{Z}^k$ containing the combinations of values (tuples) $\tau \in \mathbb{Z}^k$ that satisfy c. In the context of a CSP, where each variable v_{ij} contains an associated domain d_{ij}, the constraint propagation of c represents the inference process of removing the concrete values from d_{ij} precluding c from being satisfied. For example, in the following CSP $([v_1, v_2], [v_1 \text{ in } 1..10, v_2 \text{ in } 1..10], [|v_1 - v_2| > 5])$, the constraint propagation of $|v_1 - v_2| > 5$ removes the values 5 and 6 from the domain of v_1 and v_2, as, by taking any of these values, the constraint can never be

satisfied.

Extending this process to a CSP with $C \equiv [c_1, \ldots, c_m]$, the constraint propagation of the network is the process of propagating each $c \in C$ as many times as needed, until the CSP is found unfeasible (because the domain of a variable v_{ij} becomes empty) or the CSP is found stable (because the domains of its variables are no longer pruned by propagating the whole constraint set).

More specifically, the propagation of c (pruning the domains of $[v_{i1}, \ldots, v_{ik}]$) is seen from two different perspectives: The *local consistency* and the *rule iteration*. The former defines the local property the domains of $[v_{i1}, \ldots, v_{ik}]$ hold after propagating c. The notion of 'local' is due to the fact that each constraint c of the network is checked separately. The latter defines the concrete situations in which propagators are triggered, and the way they prune the domains of the variables.

The most used local consistency is *arc consistency*. Given c, defined on $[v_{i1}, \ldots, v_{ik}]$ (with concrete domains $[d_{i1}, \ldots, d_{ik}]$), arc consistency iterates on each domain value d_{ijt} of each variable v_{ij}, checking that there is a concrete valuation τ (mapping of variables to singleton values of their domains, with v_{ij} obviously assigned to d_{ijt}) satisfying c. Otherwise, the value d_{ijt} is removed from v_{ij}. Thus, arc consistency algorithms are asked to return the arc consistent closure of a network, that is, a sub-domain being arc consistent, and such that any larger sub-domain is not arc consistent. For example, in the following CSP $([v_1, v_2, v_3], [v_1 \text{ in } 1..3, v_2 \text{ in } 1..3, v_3 \text{ in } 1..3], [v_1 = v_2, v_2 < v_3])$, the propagation of $v_1 = v_2$ leads to no domain pruning, as $\tau_1 \equiv (v_1 = 1, v_2 = 1)$, $\tau_2 \equiv (v_1 = 2, v_2 = 2)$ and $\tau_3 \equiv (v_1 = 3, v_2 = 3)$ are (respectively) found for values 1, 2 and 3 of both v_1 and v_2. Then, the propagation of $v_2 < v_3$ prunes the value 3 from v_2 and 1 from v_3, as no feasible τ can be found for them. As the domain of v_2 has been modified, the constraint $v_1 = v_2$ is propagated again, as it includes v_2. In this case, the value 3 of v_1 is pruned, as no feasible τ can be found for it. As v_1 is involved in no more constraints, the propagation of the constraint networks successfully finishes, finding the new stable CSP $([v_1, v_2, v_3], [v_1 \text{ in } 1..2, v_2 \text{ in } 1..2, v_3 \text{ in } 2..3], [v_1 = v_2, v_2 < v_3])$.

The most well-known algorithm for ensuring arc consistency is AC3 [127]. It is presented in Figure 2.3, with $D(x_i)$ representing the domain of variable x_i, $X(c)$ representing the sequence of variables $[v_{i1}, \ldots, v_{ik}]$ involved in a constraint c, and Q being the queue of pairs (variable, constraint) waiting to be propagated. Whereas the function AC3 enqueues each pair (variable, constraint) until no more domain values are removed, the function `Revise3` is in charge of finding the satisfiable τ for each pair.

Unfortunately, arc consistency algorithms require an expensive processing. Considering the constraints on their extensional form, AC3 runs in $\Theta(er^3d^{r+1})$ time (where e is the number of constraints, r the largest arity of a constraint and d the largest domain size) and $\Theta(er)$ space. Multiple improvements to AC3 have been performed, included in the algorithms AC4 [74], AC6 [27], AC7 [28] and AC2001 [29]. However, the best complexity that can be achieved for an algorithm enforcing arc consistency on a network with any kind of constraints is in $\Theta(erd^r)$. Thus, more relaxed local consistencies

```
function Revise3(in x_i: variable; c: constraint): Boolean ;
  begin
1     CHANGE ← false;
2     foreach v_i ∈ D(x_i) do
3         if ∄τ ∈ c ∩ π_{X(c)}(D) with τ[x_i] = v_i then
4             remove v_i from D(x_i);
5             CHANGE ← true;
6     return CHANGE ;
  end

function AC3/GAC3(in X: set): Boolean ;
  begin
      /* initalisation */;
7     Q ← {(x_i, c) | c ∈ C, x_i ∈ X(c)};
      /* propagation */;
8     while Q ≠ ∅ do
9         select and remove (x_i, c) from Q;
10        if Revise(x_i, c) then
11            if D(x_i) = ∅ then return false ;
12            else Q ← Q ∪ {(x_j, c') | c' ∈ C ∧ c' ≠ c ∧ x_i, x_j ∈ X(c') ∧ j ≠ i};
13    return true ;
  end
```

Figure 2.3: Algorithm AC3

have been presented, pruning less the variable domains, but running the propagation algorithm faster.

In Figure 2.3, it can be seen that there are two sources of computational effort that can be relaxed. The first one is the *events* triggering a pair (variable, constraint) to be enqueued for constraint propagation. In the proposed algorithm, the pruning of a single value in variable v_i (due to constraint c) directly enqueues any pair (v_j, c') for any constraint c' (different from c) involving both to v_i and v_j. For example, in the CSP proposed before, the pruning of value 3 from the domain of v_2 (due to the constraint $v_2 < v_3$), enqueues the pair (v_1, $v_1 = v_2$). Whereas this is very convenient (as the further propagation of $v_1 = v_2$ prunes the value 3 from v_1), it is not as clear on the other way round. That is, regarding $v_2 < v_3$, once it performed propagation once, the only events that would trigger the pruning of new variable domain values are: Increasing the minimum value of v_2 (or bound its domain to a singleton), and decreasing the maximum value of v_3 (or bound its domain to a singleton). Thus, the constraint $v_2 < v_3$ should be only enqueued when any other constraint of the network (in this case, just $v_1 = v_2$) matches one of these events.

Constraint solvers usually recognize different events, as: $onDomain$ (when a value is removed from the domain of a variable), $onBound$ (when the value being removed

is either the lower or upper bound of the domain) and $onValue$ (when the domain of the variable is bound to a singleton). These solvers might allow their different kinds of primitive constraints to be parameterizable on the kind of events that enqueue them (as well as to allow the specification of events that will enqueue any new constraint being programmed by the user).

The second source of computational effort that can be relaxed is the local consistency ensured for each pair (v_i, c) being checked (with $[v_{i1}, \ldots, v_{ik}]$ being the sequence of variables involved on c). A first option, called $bound(D)$ consistency, is to look for a feasible τ only for the bounds (minimum and maximum value) of each v_{ij}. A second option, called $range$ consistency, is to still check any domain value d_{ijt} of each variable v_{ij}, but, when looking for an associated τ, consider the domain of the remaining variables $[v_{i1}, \ldots, v_{ik}]$ as if they actually contained all the values in the range from their minimum and maximum values). A third option, called $bound(Z)$ consistency, is to combine the first and the second options. Constraint solvers might allow their different kind of primitive constraints to be parameterizable on the local consistency algorithm they use, as well as defining different local consistency algorithms for any new constraint being programmed by the user.

Anyway, arc consistency is strictly stronger than $range$ and $bound(D)$ consistencies, which are themselves strictly stronger than $bound(Z)$ consistency. Regarding $bound(D)$ and $range$, they are incomparable. For example, the following CSP $([v_1, v_2, v_3, v_4, v_5, v_6]$, $[v_1$ in $\{1,2\}, v_2$ in $\{1,2\}, v_3$ in $\{2,3,5,6\}, v_4$ in $\{2,3,5,6\}, v_5$ in $\{5\}, v_6$ in $\{3,4,5,6,7\}]$, $[all\_different(v_1, v_2, v_3, v_4, v_5, v_6)])$ constrains its six variables to be pairwise different. However, instead of using multiple binary constraints (as in Sections 2.1.1 and 2.1.2), it uses the well-know $all\_different$ global constraint. Global constraints are specialized constraints containing a parameterizable number of variables. They encapsulate a set of other (possibly primitive) constraints, and can be associated with more powerful consistency algorithms, as they can take into account the simultaneous presence of simple constraints to further reduce the domains of the variables [154].

In the particular example, the propagation achieved by each local consistency is:

- $bound(Z)$: $D \equiv [v_1$ in $\{1,2\}, v_2$ in $\{1,2\}, v_3$ in $\{3,5,6\}, v_4$ in $\{3,5,6\}, v_5$ in $\{5\}, v_6$ in $\{3,4,5,6,7\}]$.
- $range$: $D \equiv [v_1$ in $\{1,2\}, v_2$ in $\{1,2\}, v_3$ in $\{3,6\}, v_4$ in $\{3,6\}, v_5$ in $\{5\}, v_6$ in $\{3,4,6,7\}]$.
- $bound(D)$: $D \equiv [v_1$ in $\{1,2\}, v_2$ in $\{1,2\}, v_3$ in $\{3,5,6\}, v_4$ in $\{3,5,6\}, v_5$ in $\{5\}, v_6$ in $\{4,5,6,7\}]$.
- Arc consistency: $D \equiv [v_1$ in $\{1,2\}, v_2$ in $\{1,2\}, v_3$ in $\{3,6\}, v_4$ in $\{3,6\}, v_5$ in $\{5\}, v_6$ in $\{4,7\}]$.

2.2.2 Search Exploration

Local consistency prunes the variable domains, ensuring that a value being removed does not participate in any solution to the problem. However, not all values that remain in the domains necessarily are part of some solution. For example, applying arc consistency to Golomb-5, the variable domains are pruned to: $[m_0$ in 0, m_1 in $1..8$, m_2 in $3..12$, m_3 in $6..13$, m_4 in $10..15$, $d_{1,0}$ in $1..8$, $d_{2,0}$ in $3..12$, $d_{3,0}$ in $6..13$, $d_{4,0}$ in $10..15$, $d_{2,1}$ in $1..11$, $d_{3,1}$ in $3..12$, $d_{4,1}$ in $6..14$, $d_{3,2}$ in $1..10$, $d_{4,2}$ in $3..12$, $d_{4,3}$ in $2..9]$. The search space S has been reduced to 11,678,515,200,000 candidates (a 0.00001% of the initial 1,152,921,504,606,846,976). However, it is still very far away from finding the optimal solution (or, at least, any of the 82 feasible ones). Moreover, for Queens-4, applying arc consistency does not prune any of the 256 initial candidates of S. Thus, the solving of a CSP or a COP requires a search algorithm, looking for valid assignments.

CP($\mathcal{FD}$) provides both systematic *backtracking search* algorithms, and non-systematic *local search* ones. The former guarantees that a solution will be found if one exists, and can be used to show that a problem does not have a solution. The latter are stochastic-based, and thus do neither guarantee an optimal solution, nor can be used to show a CSP does not have a solution. However, sometimes such algorithms are more effective (with better solving performance) at finding a solution if one exists, as well as at finding a good approximation to an optimal solution. In any case, in this thesis only systematic search is considered.

A backtracking search can be seen as performing a traversal of a search tree (e.g. a depth first traversal). It is generated as the search progresses, and represents alternative choices that may have to be examined in order to find a solution. Extending a node in the tree is often called a branching strategy. When a node is generated, its exploration is used to check whether it leads to a solution, and also to prune subtrees containing no solutions.

More specifically, a node p, placed in the *j*-th level of the search tree, can be seen as a new CSP' $\equiv (V, D', C')$. It is similar to the original CSP $\equiv (V, D, C)$ being solved, but $C' \equiv C \cup \{b_1, \ldots, b_j\}$ enlarges the original constraint network with a set of *branching constraints*, where each b_i is the constraint posted at level i in the search tree. Extend a node p (to generate the next tree level) is done by generating new nodes $p_1, \ldots, p_k$, where each p_i is a new CSP'' $\equiv (V, D'', C'')$, with $C'' \equiv C' \cup \{b_{j+1\_i}\}$. To ensure completeness, the branching constraints $[b_{j+1\_1}, \ldots, b_{j+1\_k}]$ respectively added to $p_1, \ldots, p_k$ must be mutually exclusive and exhaustive. Without loss of generality, in this thesis only unary branching constraints are considered, which can be classified into: *Enumeration*, where variable v_i is instantiated in turn to each value in its domain. *Binary choice points*, where left path $b_{j+1\_1} \equiv v_i = a$, and right path $b_{j+1\_2} \equiv v_i$!= a (in [106] it was shown that this branching technique is exponentially more powerful than the enumeration one). *Domain splitting*, where $b_{j+1\_1} \equiv v_i <= a$ and $b_{j+1\_2} \equiv v_i > a$.

By considering each node p as a CSP, its exploration just consists of maintaining a

local consistency property on it. For example, the arc consistency constraint propagation for Golomb-5 and Queens-4 presented before (roughly following the algorithm of Figure 2.3) can be seen as the propagation achieved in the root tree node. However, for the rest of the search, only small changes occur between successive calls to the constraint propagation algorithm (i.e., for any node of the tree, the difference between its C' and the one of its father C is just the last branching constraint being posted). Thus, the computational effort for maintaining local consistency on each node becomes cheaper. In any case, as described in Section 2.2.1, different consistency levels can be used, with the corresponding tradeoff between time and pruning. In general, it is commonly accepted that the local consistency level chosen depends on the concrete CSP, and that different choices may be made for different constraints within a same CSP.

The constraint propagation of a node p leads to three possible situations:

1. The CSP becomes stable, and all its variables are bound to singleton domains. Then, a solution has been found, and the search exploration stops.
2. The CSP becomes stable, but its search space S still contains different candidates. Then, the search exploration continues, extending the node by generating its children $[n_1, \ldots, n_k]$. An unistantiated variable v_a is selected, posting the associated branching constraint b_i to each n_i being created.
3. The CSP becomes unfeasible. Then, a *fail* has been found, and the subtree of p is not even generated for its exploration. The next node (once again, performing a depth first traverse of the tree) is selected to be explored.

When solving a COP, a few modifications must be done in the scheme just presented. Assuming that the cost function F has been assigned to variable v_i, a common approach is to explore the tree implementing a constraint-based version of branch-and-bound [100]. On it, a backtracking search is used to find a solution s (for which v_i takes value val). Then, instead of stopping, the search continues, but an additional constraint $v_i < val$ is added to the network, enforcing further solutions to improve the achieved bound for the cost function. This process is repeated until the resulting CSP is unsatisfiable, in which case the last solution is said to be the optimal one.

Finally, besides domain consistency, another efficiency factor being considered in this thesis is the variable and value selection heuristics. They determine the shape of the tree, and there are many case studies showing the performance impact of selecting a suitable variable and value ordering [79], [83]. A variable ordering can be either static, where the ordering is fixed and determined prior to search, or dynamic, where the ordering is determined as the search progresses. Some examples of fixed selection criterion are textual order or most/less constrained variable. Some examples of dynamic criterion are smallest/largest domain (also known as *first fail* [94]), smallest/largest minimum/maximum value, smallest/largest difference between the two smallest/largest values or random selection. Regarding value ordering, some selection criterion are increasing/decreasing order, mean, median and random selection.

Also, the variable and value selection heuristics are relevant to encourage or discourage the use of symmetry breaking constraints, as the ones used in the Queens and Golomb problems. That is, breaking the symmetries of a problem reduces the search space of a problem. However, picking out particular solutions in each symmetry class may conflict with the tree exploration order followed by the variable and value selection heuristics [198].

To finish the section, two examples are used, showing the performance impact of the variable/value heuristics and the local consistency algorithms being used, respectively.

First, the variant Queens-6 of the CSP of Figure 2.1 (using $all\_different$ global constraints and a $v_1 < v_n$ symmetry breaking constraint) is solved. Figures 2.4 and 2.5 present the search exploration for achieving a first feasible solution. The former labels the variables $[v_1, \ldots, v_n]$ in their textual order (selecting the domain values of each variable in an increasing order). The latter labels the variables selecting first the variable v_i with less domain values (selecting first the greatest value not greater than the median). As it can be seen, the search of Figure 2.5 just needs to explore 7 nodes to find the solution (achieving just 1 fail). The constraints posted in the solution path (nodes $[2, 3, 4, 5, 7]$) are $[v_1 = 4,\ v_8 = 6,\ v_3 = 5,\ v_5 = 1,\ v_6 = 7]$, leading to the solution $[4, 8, 5, 3, 1, 7, 2, 6]$. On the other hand, the search of Figure 2.4 needs to explore 49 nodes to find the solution (achieving 23 fails). The constraints posted in the solution path (nodes $[2, 16, 36, 37, 41, 45, 49]$) are $[v_1 = 1,\ v_2 \mathrel{!}= 3,\ v_2 \mathrel{!}= 4,\ v_2 = 5,\ v_3 \mathrel{!}= 2,\ v_3 \mathrel{!}= 7,\ v_4 \mathrel{!}= 2]$, leading to the solution $[1, 5, 8, 6, 3, 7, 2, 4]$. As it can be seen, some wrong decisions lead to explore subtrees containing no solutions: $v_2 = 3$, subtree with root node 3, 13 nodes. $v_2 = 4$, subtree with root node 17, 19 nodes. $v_3 = 2$, subtree with root node 38, 3 nodes. $v_3 = 7$, subtree with root node 42, 3 nodes. $v_4 = 2$, subtree with root node 46, 3 nodes.

Second, Golomb-5 (COP of Figure 2.2, but using an $all\_different$ global constraint) is solved. Figures 2.6 and 2.7 present the search exploration for achieving the optimal solution. The former applies value consistency for the $all\_different$ constraint, which treats it as a sequence of pairwise binary disequality constraints. The latter applies arc consistency for the $all\_different$ constraint. As it can be seen, whereas the search of Figure 2.7 just needs to explore 11 nodes to find the solution (achieving just 4 fails), the search of Figure 2.6 explores 21 nodes (achieving 9 fails). Both search explorations find a first feasible solution in the path $[2, 3, 4, 5]$, by posting $[m_1 = 1,\ m_2 = 3,\ m_3 = 7,\ m_4 = 12]$. At this solution node 6, the constraint $m_4 < 12$ is added to the network. Then, the search continues, and a new solution $[m_1 = 1,\ m_2 = 4,\ m_3 = 9,\ m_4 = 11]$ is found. The constraint $m_4 < 11$ is added, and the search concludes without finding any new solution, turning $[0, 1, 4, 9, 11]$ into the optimal one.

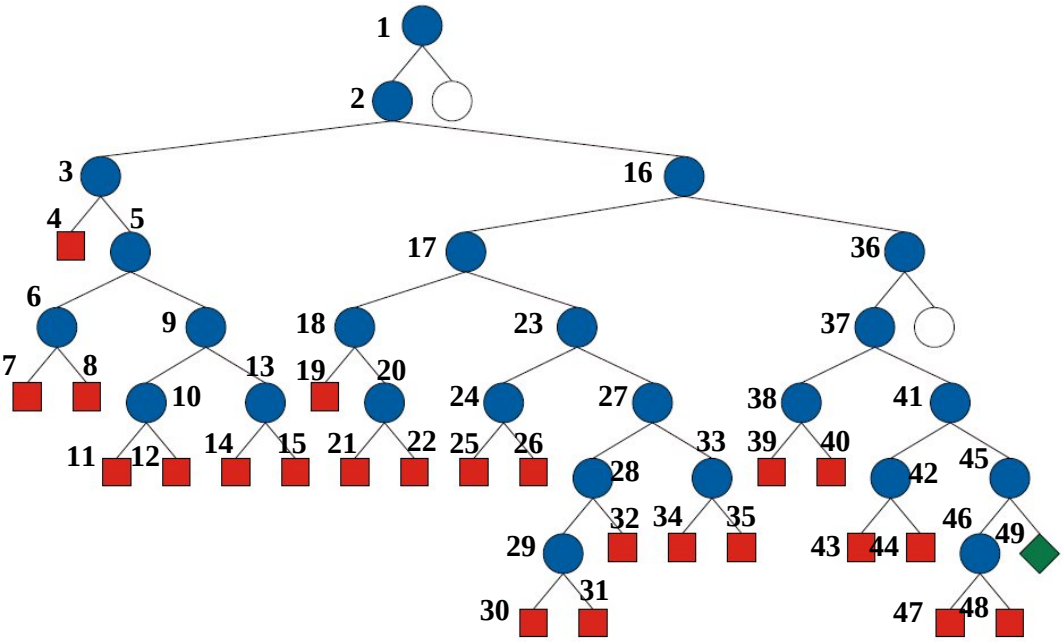

Figure 2.4: Queens-6 with Lexicographic Variable and Value Order

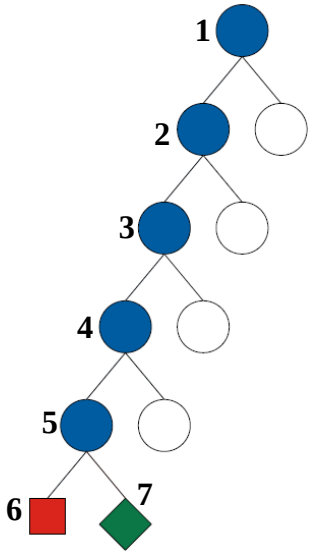

Figure 2.5: Queens-6 with First Fail and Medium Values First

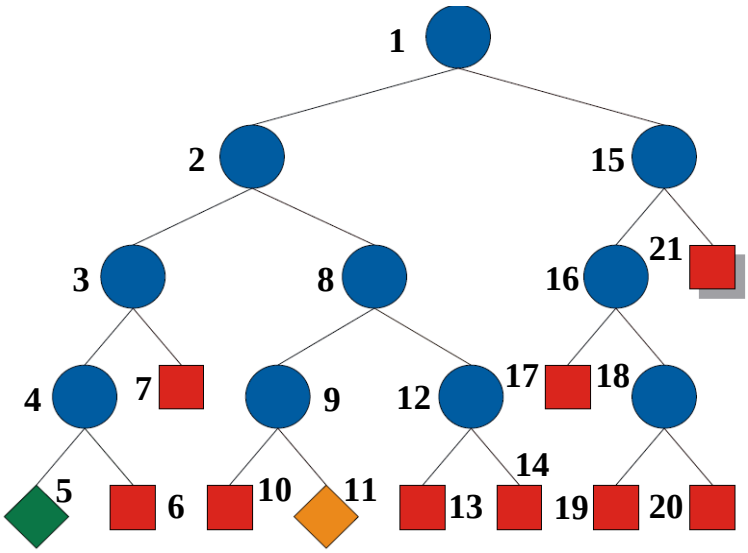

Figure 2.6: Golomb-5 with $all\_different$ Value Consistency

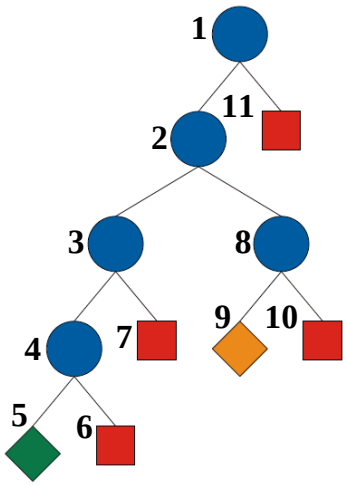

Figure 2.7: Golomb-5 with $all\_different$ Arc Consistency

2.3 CP($\mathcal{FD}$) Modeling Paradigms

This section presents an introduction to the different modeling paradigms available in CP($\mathcal{FD}$). It provides a brief description of each of the state-of-the-art systems considered in the third part of the research of the thesis. It justifies the system set being selected, with the aim of covering the different modeling features available, but relying on a minimal set of different constraint solver libraries. Moreover, the use of two systems per paradigm allows to widen the conclusions for a paradigm.

2.3.1 Algebraic CP(FD)

The first modeling paradigm being considered is Algebraic CP($\mathcal{FD}$). It sets a context for high-level specific purpose languages based on algebraic formulations. It abstracts the CSP or COP specification from the concrete solver or algorithm used to solve it. It provides a high expressivity to the user, including the use of basic constraints (which can be combined to obtain complex ones), new constraint definitions (via predicates), combinators, enumerated types, array and set data structures, predefined and user-defined search strategies, and an isolation from the general model to the instance-dependent input data. The abstraction from the model to the solver allows to try the same model over different solvers with no extra effort. However, the use of specific purpose languages makes more difficult the modeling of constraint-independent components (i.e., the tasks that are not strictly constraint modeling), as well as the development of non-monolithic problems and the integration of the model into larger applications.

Some algebraic CP($\mathcal{FD}$) systems are MiniZinc [142], IBM ILOG OPL [193], Comet [60], Minion [81] and Numberjack [98]. To position $\mathcal{TOY}(\mathcal{FD})$ w.r.t. algebraic CP($\mathcal{FD}$), the state-of-the-art systems MiniZinc and ILOG OPL are considered. MiniZinc has been selected because it is high-level enough to express most constraint problems easily, but low-level enough that it can be mapped onto existing solvers easily and consistently. To do so, the MiniZinc interpreter includes FlatZinc, a low-level solver input language that is the target language for MiniZinc. It is designed to be easy to translate into the form required by a solver. In fact, any MiniZinc targeted-solver can customize its own global constraint compilation from MiniZinc to FlatZinc. Currently, MiniZinc might be seen as the standard algebraic CP($\mathcal{FD}$) modeling language. Most state-of-the-art constraint solver libraries include an interface to FlatZinc, and the MiniZinc challenge [7] is organized every year to compare various constraint solvers on the same problems sets, as well as to build up a library of interesting problem models.

ILOG OPL has been selected because it is an industrial market leader, being the modeling language selected by most of the companies for CSP and COP solving during the last decade. It can be seen as the pioneering algebraic CP($\mathcal{FD}$) language, from which the rest of languages have departed from. Thanks to the IBM academic initiative [107] these products are free for academic purposes. ILOG OPL is not as much generic

as MiniZinc is, in the sense that an OPL model can only be targeted to the different constraint solvers that ILOG provides.

2.3.2 Object Oriented CP($\mathcal{FD}$)

The second modeling paradigm being considered is Object Oriented (OO) CP($\mathcal{FD}$). As the need for solving CSP's and COP's increased, the industry requested new general-purpose modeling languages. The aim was to integrate the CP($\mathcal{FD}$) techniques into other programming communities, also making possible the coexistence of CP($\mathcal{FD}$) models and other constraint independent components within larger applications. Thus, the integration of CP($\mathcal{FD}$) into the OO paradigm (and in concrete into the C++ language) was quite natural and interesting, both in terms of the problem specification and its solving. First, the high efficiency of C++ [36] allows the CP($\mathcal{FD}$) libraries implemented in this language to obtain a higher solving efficiency. Second, as now this libraries must provide a C++ API, users can take advantage of features such as the abstraction, encapsulation, inheritance and polymorphism [33]. Modeling using the C++ API interface of a CP($\mathcal{FD}$) library provides a tradeoff between efficiency and abstraction, as models become solver-specific (and thus more efficient), but the user has to explicitly manage all these constraint elements that algebraic CP($\mathcal{FD}$) languages already abstracted.

Some C++ CP($\mathcal{FD}$) libraries are Gecode [78], IBM ILOG Solver [12] and Cacao [41]. Also, due to the new standard JSR 331 [112], some Java CP($\mathcal{FD}$) libraries have been developed, with JaCoP [110], Choco [54] and YACS [206] as some examples. To position $\mathcal{TOY}(\mathcal{FD})$ w.r.t. OO CP($\mathcal{FD}$), the C++ CP($\mathcal{FD}$) state-of-the-art systems Gecode and ILOG Solver are considered.

Gecode has been selected because it is a free software CP($\mathcal{FD}$) library with state-of-the-art performance (awarded with the best results in the 2008-2012 editions of the MiniZinc Challenge). Relying on an exhaustive documentation and with an alive community (which goes far beyond the education training), it provides a wide set of templates allowing users to be first class citizens in the development of new variable domains, constraints and search procedures (which follow exactly the very same techniques that Gecode itself provides). Besides that, Gecode interfaces FlatZinc, as it is thus possible to make a modeling and solving comparison between programming in MiniZinc (interfaced to Gecode) and directly in Gecode.

ILOG Solver has been selected because it is the constraint solver targeted by OPL when solving generic $\mathcal{FD}$ problems. It belongs to the ILOG CP package, which contains the ILOG Concert modeling library and two other solver libraries for specific scheduling and routing problems. As it happened with ILOG OPL (where a translation to ILOG Solver was done), modeling with ILOG Concert also needs a translation to ILOG Solver (in the context of C++ CP($\mathcal{FD}$) programming). Thus, a modeling and solving comparison between programming in ILOG OPL and ILOG Concert (and thus between both translations to ILOG Solver) turns to be quite interesting, as well as to compare this gap with

the one produced between MiniZinc (against Gecode) and Gecode.

2.3.3 Logic Programming CP($\mathcal{FD}$)

The integration of CP($\mathcal{FD}$) techniques into declarative languages has also been quite interesting from an industrial point of view. These languages offer a higher abstraction level for programming, easing the development of secure and maintainable programs [121]. They are based on an strong formalism, with a clear distinction between its declarative and operational (execution model) semantics. Thus, the paradigm of Logic Programming (LP) [120] has been used in a vast range of application domains. Whereas the declarative semantics of LP treats programs as logic theories based on a set of (possibly non-deterministic) clauses, its operational semantics is based in the SLD calculus of Robinson [160] for solving logic goals via unification.

The integration of CP($\mathcal{FD}$) into LP sets up the framework for the paradigm of Constraint Logic Programming over Finite Domains (CLP($\mathcal{FD}$)), which is the third modeling paradigm being considered. On the one hand, in terms of CSP and COP modeling, the new paradigm takes most of the advantages of both algebraic and C++ languages: First, it abstracts the model from the concrete solver. This allows to plug-in the same model over different solvers, as in algebraic languages. This also allows to avoid explicit handling of a constraint solver object (managing the control of its decision variables, constraints, objective function, constraint store, constraint propagation, search engine, search control, as well as the garbage collection of all these elements), as OO must do. Second, it provides high expressivity, even enhancing the one of algebraic and C++ languages by adding logic features such as relational notation, non-determinism, backtracking, logical variables, domain variables, and the capability of reasoning with models. Third, by using a logic general-purpose programming language, it also has the same capabilities of C++ languages for modeling constraint-independent components, tackling non-monolithic models and integrate them into larger applications. On the other hand, in terms of CSP and COP solving, CLP($\mathcal{FD}$) systems decrease the C++ CP($\mathcal{FD}$) efficiency, as they must intermix constraint solving with its SLD resolution operational semantics (an inherent overhead comes from interfacing the constraint solver to the system and coordinating it). To alleviate the effects of this overhead, and make the interface to the constraint solver more clean and elegant, most CLP($\mathcal{FD}$) systems have implemented its own host solver, attached to the underlying system architecture.

Some CLP($\mathcal{FD}$) systems are SICStus Prolog [178], SWI-Prolog [190], GNU Prolog [5], Ciao Prolog [4], B-Prolog [31] and ECLiPSe [16], all of them using Prolog as host language (and the latter interfacing Gecode for $\mathcal{FD}$ constraint solving). To position $\mathcal{TOY}(\mathcal{FD})$ w.r.t. CLP($\mathcal{FD}$), the state-of-the-art systems SICStus Prolog and SWI-Prolog are considered.

SICStus Prolog has been selected because it is an ISO standard compliant, Prolog development system with state-of-the-art performance [71]. It is built around a

high performance Prolog engine that can use the full virtual memory space, and is efficient and robust for large amounts of data and large applications. Moreover, SICStus is the host language of the CFLP($\mathcal{FD}$) systems being considered. Thus, a comparison between their modeling and solving performance and the one of SICStus turns to be quite interesting. Also, SICStus provides a Prolog-C++ (also Prolog-Java) communication framework allowing to define a Prolog predicate prototype whose implementation is contained in a C++ function. This includes a conversion between Prolog and C++ parameters, which also contains a C++ representation of Prolog terms.

SWI-Prolog has been selected because it is an efficient free software CLP($\mathcal{FD}$) system, that has been driven by the needs for real-world applications. It is widely used in research and education, as well as for many commercial applications [203]. Thus, a modeling and solving comparison between SICStus and SWI-Prolog turns to be quite interesting.

2.3.4 Functional Programming CP($\mathcal{FD}$)

The paradigm of Functional Programming (FP) [128] sets the context for a second family of declarative languages. Whereas the declarative semantics of FP treats programs as theories based on a set of deterministic functions, its operational semantics is based in reducing expressions via rewriting [20]. The integration of CP($\mathcal{FD}$) into FP sets up the framework for the paradigm of Constraint Functional Programming over Finite Domains (CFP($\mathcal{FD}$)). In terms of CSP and COP modeling, the three advantages presented for CLP($\mathcal{FD}$) languages (w.r.t. algebraic and C++ ones) also apply to CFP($\mathcal{FD}$), where the higher expressivity of the paradigm comes from the addition of functional features such as functional notation, curried expressions, higher-order functions, patters, partial application, lazy evaluation and the use of types and polymorphism. Also, as functions are first-class citizens, they can be used as any other object in the language (i.e., results, input arguments and elements of data structures), giving CFP($\mathcal{FD}$) a greater flexibility than CLP($\mathcal{FD}$). In terms of solving, FP provides efficient and (under particular conditions) optimal evaluation strategies using demand-driven evaluation. However, as in CLP($\mathcal{FD}$), intermixing constraint solving with the rewriting-based operational semantics implies an inherent overhead for CFP($\mathcal{FD}$) performance. Thus, the integration of CP($\mathcal{FD}$) techniques into FP seems to be at least as interesting as in LP. However, the literature lacks of mature proposals in this sense.

Some FP CP($\mathcal{FD}$) libraries are Alice ML [10] and FaCiLe [69]. On the one hand, they provide some functional features, such as higher-order functions, type inference, strong typing, user-defined constraints and the manipulation of potentially infinite data structures by using explicit delayed expressions (lazyness is not an inherent characteristic of their resolution mechanism). However, they do not provide general-purpose functional programming languages, as they are based on a low-level and imperative API.

More recently [169] developed a Monadic Constraint Programming (MCP) framework, integrating CP($\mathcal{FD}$) techniques in the pure functional programming language Haskell [148]. It illustrates how the abstractions and mechanisms from FP such as monads, higher-order functions, continuations and lazy evaluation are valuable notions for defining and building CP($\mathcal{FD}$) systems. The FD-MCP framework [205] implements an embedded domain-specific language in Haskell for constraint programming, abstracting the concrete CP($\mathcal{FD}$) solver being used. They also focus in solving performance, providing an interface to Gecode, which can be accessed either directly from the system (intermixing constraint solving with rewriting), or by compiling the Haskell model to a C++ Gecode one. Thus, the use of FD-MCP seems to be quite interesting. Unfortunately, the system seems to be at a beta version, as it does not provide neither a user nor a reference manual to learn how to use it. Besides that, none of the provided examples do compile in the recent versions of Gecode (they do not mention the Gecode version they are using), and some of them directly do not even generate any C++ code. For these reasons the use of FP CP($\mathcal{FD}$) systems is not considered in this thesis, although they provide some strengths for CSP and COP modeling and solving.

2.3.5 Multi-paradigm Declarative Programming CP($\mathcal{FD}$)

Each single declarative paradigm provides relevant programming features. FP emphasizes generic programming, using higher-order functions and polymorphic typing. Also, the use of lazy or demand-driven computation strategies provide efficient and, under particular conditions, optimal evaluation of expressions. LP supports the computation with partial information (logic variables) and the non-deterministic search for solutions (with model reasoning implemented via non-deterministic predicates and backtracking). Thus, the integration of both paradigms into a Functional Logic Programming (FLP) [161], [91], [19] framework obtains both the higher flexibility of LP and the higher efficiency of FP. Moreover, all the good properties achieved on the integration of CP($\mathcal{FD}$) into LP also holds for FLP, setting up the framework for the paradigm of Constraint Functional Logic Programming over Finite Domains (CFLP($\mathcal{FD}$)).

In terms of modeling, CFLP($\mathcal{FD}$) represent one of the most complete approaches within CP($\mathcal{FD}$) systems. First, its declarative nature abstracts the problem specification, allowing the user to model a problem by simply describing the properties its solutions must hold. This represents an advantage w.r.t. imperative languages (as the ones of C++ CP($\mathcal{FD}$) systems), where the way the model is built is procedural (although the paradigm is still declarative, i.e., the C++ objects are used to describe a declarative model). Second, its general purpose nature allows the integration of the model into larger applications, in contrast to specific-purpose languages (as the ones of algebraic CP($\mathcal{FD}$) systems). Third, as mentioned before, its high expressiveness includes the main features from both LP and FP languages (as the ones of CLP($\mathcal{FD}$) and CFP($\mathcal{FD}$) systems, respectively). The logic component includes features such as relational no-

tation, non-determinism, backtracking, logical variables, domain variables and the capability of reasoning with models. The functional component includes features such as functional notation, curried expressions, higher-order functions, patterns, partial applications, lazy evaluation, types, polymorphism and constraint composition.

In terms of solving, CFLP($\mathcal{FD}$) has the same kind of overhead as CLP($\mathcal{FD}$), i.e., coming from intermixing constraint solving with the operational semantics (an inherent overhead comes from interfacing the constraint solver to the system and coordinating it). Unfortunately, CFLP($\mathcal{FD}$) cannot decrease this inherent overhead (as CFP($\mathcal{FD}$) can do via compiling the native language model to a host constraint solver language one). The problem comes from the logic feature of reasoning with models. With it, the system can retract from the constraint store some of the regular constraints of the model (not to mix them up with the constraints retracted due to the inherent backtracking associated to a search exploration). Thus, any CFLP($\mathcal{FD}$) native model must be executed on its native framework.

A first approach to set up a CFLP($\mathcal{FD}$) framework consists of extending a logic language with FP features. As functions can be considered as specific relations, the language is extended with syntactic sugar to allow functional notation. Two systems of this family are HAL [62] and Oz [194]. The former is explicitly designed to support the experimentation with different constraint solvers and the development of new ones. It allows to call solvers written in other languages (e.g. C) with little overhead, as well as to "plug and play" with different constraint solvers over the same domain. The latter generalizes the CLP($\mathcal{FD}$) and concurrent constraint programming paradigms, including first-class functions and constraints (which are also treated as functions). However, both HAL and Oz rely on strict operational semantics that do not support optimal evaluation as in FP.

A second approach to set up a CFLP($\mathcal{FD}$) framework consists of extending a functional language with LP features. It combines the SLD resolution mechanism of LP with the lazy or demand-driven function evaluation of FP. Two systems of this family are PAKCS [93] (one of the available system versions of the Curry [90] language) and $\mathcal{TOY}(\mathcal{FD})$ [72], [124], [40].

Regarding their declarative semantics, PAKCS and $\mathcal{TOY}(\mathcal{FD})$ programs have a syntax mostly borrowed from Haskell, and can be seen as theories based on a set of functions, possibly higher order, non-deterministic and with conditional rules. Both systems rely on a Constructor-based Rewriting Logic: CRWL [85]. It provides a basis for FLP with non-strict functions (not all its arguments are needed to be evaluated to apply a function rule) non-deterministic functions (with possibly several reductions for given, even ground, arguments) and call-time choice (arguments are shared to avoid multiple evaluations of the same subterm).

Regarding their operational semantics, whereas $\mathcal{TOY}(\mathcal{FD})$ implements *lazy narrowing*, PAKCS integrates it with *residuation* [91]. The execution model of residuation suspends function calls until they are instantiated enough for deterministic reduction.

Thus, in this setting, non-determinism must be expressed by explicit disjunctions. Narrowing overcomes this restriction, allowing to apply the rule of a function with unknown arguments by instantiating them. As it can be seen, narrowing combines the pure FP mechanism of rewriting with the pure LP mechanism of unification. On the operational side, the LP resolution principle must be extended to deal with replacements of subterms, so that a function rule is unified (instead of rewritten) with the subterm under evaluation.

Under the condition that the function rules of a program form an *Inductively Sequential Term Rewriting System* [20], a lazy narrowing strategy is ensured to be optimal (i.e., performing only the strictly necessary narrowing steps to compute a result). The discrimination between needed and non-needed narrowing steps is given by the notion of *definitional trees* [18]. In $\mathcal{TOY}(\mathcal{FD})$, the operational semantics is based on a Constraint Lazy Narrowing Calculus: CLNC($\mathcal{FD}$) [123], which relies on the use of definitional trees to implement lazy narrowing for goal solving.

Finally, regarding implementation, both $\mathcal{TOY}(\mathcal{FD})$ and PAKCS are implemented in SICStus Prolog. There are several approaches to compile FLP languages with demand-driven evaluation strategies into Prolog (relying on the Prolog backtracking mechanism to implement non-deterministic search) [122], [52]. Whereas narrowing strategies can be compiled into pure Prolog, residuation demands for coroutining. Thus, instead of using an abstract machine for running byte-code or intermediate code from compiled programs, both $\mathcal{TOY}(\mathcal{FD})$ and PAKCS use SICStus Prolog as their object language (which includes SICStus `clpfd` for $\mathcal{FD}$ constraint solving).

In summary, the state-of-the-art CFLP($\mathcal{FD}$) systems $\mathcal{TOY}(\mathcal{FD})$ and PAKCS purely integrate the FP and LP paradigms, both in their language expressivity and in their solving mechanisms. They rely on the same declarative and operational semantics, besides being implemented in the same language. Thus, PAKCS is the system being considered to position $\mathcal{TOY}(\mathcal{FD})$ w.r.t. the CFLP($\mathcal{FD}$) paradigm itself. Moreover, besides a comparison between the PAKCS and $\mathcal{TOY}(\mathcal{FD})$ modeling and solving capabilities, a comparison with the ones of SICStus `clpfd` turns interesting.

2.4 $\mathcal{TOY}(\mathcal{FD})$

This section presents a formal definition of the $\mathcal{TOY}(\mathcal{FD})$ language, supporting the solving of syntactic equalities and disequalities (via a Herbrand solver: $\mathcal{H}$), as well as the solving of $\mathcal{FD}$ constraints (via a connected CP($\mathcal{FD}$) solver). Then, some introductory $\mathcal{TOY}(\mathcal{FD})$ examples are provided, giving a brief overview of both the modeling features and the goal solving mechanism of the system. The system is available at: `http://gpd.sip.ucm.es/ncasti/TOY(FD).zip`.

2.4.1 Language

To provide a formal definition of the $\mathcal{TOY}(\mathcal{FD})$ language, the constructor-based signatures $\Sigma = \langle DC, FS \rangle$ are used, where $DC = \bigcup_{n \in \mathbb{N}} DC^n$ and $FS = \bigcup_{n \in \mathbb{N}} FS^n$ are sets of *data constructors* and *defined function symbols* with associated arities, respectively. As notational conventions, it is assumed that $c, d \in DC$, $f, g \in FS$ and $h \in DC \cup FS$. It is also assumed that countably many variables (noted as X, Y, Z, etc.) are available. Given any set $\mathcal{X}$ of variables, $Exp_\Sigma(\mathcal{X})$ denotes the set of all terms built from symbols in $\mathcal{X} \cup DC \cup FS$, and also the set $Term_\Sigma(\mathcal{X})$ of all terms built from symbols in $\mathcal{X} \cup DC$. Whereas terms $l, r, e \in Exp_\Sigma(\mathcal{X})$ are called *expressions*, terms $s, t \in Term_\Sigma(\mathcal{X})$ are called *constructor terms* or also *data terms*. Expressions without variables are called *ground* or *closed*.

A $\mathcal{TOY}(\mathcal{FD})$ program or model consists of *datatype* declarations, *type aliases*, *infix operator* definitions and rules for defining functions. The syntax is mostly borrowed from Haskell [148], with the remarkable exception that, whereas variables begin with upper-case letters, constructor and function symbols use lower-case. In particular, functions are *curried*, and the usual conventions about associativity of application hold.

Datatype definitions define new (possibly polymorphic) *constructed types*. For example, the datatype definition `data nat = zero | suc nat` defines a constructed type for natural numbers. This definition also determines the set of data constructors for the type (in the example, there are two possible data constructors to be used when generating a `nat`).

Types $\tau, \tau', \ldots$ can be constructed types, tuples $(\tau_1, \ldots, \tau_n)$, or functional types of the form $\tau \to \tau'$. As usual, $\to$ associates to the right. The language includes predefined types, such as `[A]` (the type of polymorphic lists, for which Prolog notation is used), `bool` (with constants `true` and `false`), `int` for integer numbers, or `char` (with constants `'a','b',...`).

Each defined function $f \in FS^n$ has an associated principal type of the form $\tau_1 \to \ldots \to \tau_n \to \tau$ (where τ does not contain $\to$). As usual in functional programming, types are inferred and, optionally, can be declared in the program.

The two syntactic domains *patterns* and *expressions* must be distinguished. Patterns can be understood as denoting data values, i.e., values not subject to further evaluation, in contrast to expressions, which can be possibly reduced by means of the rules of the program. Patterns $t, s, \ldots$ are defined by $t ::= X \mid (t_1, \ldots, t_n) \mid c\ t_1 \ldots t_n \mid f\ t_1 \ldots t_n$, where $c \in DC^m$, $n \leq m$, $f \in FS^m$, $n < m$, and t_i are also patterns. Notice that partial applications (i.e., applications to less arguments than indicated by the arity) of c and f are allowed as patterns, which are then called a *higher order (HO) pattern*, because they have a functional type. Therefore, function symbols, when partially applied, behave as data constructors. HO patterns can be manipulated as any other patterns. In particular, they can be used for matching or checked for equality. With this intensional point of view, functions become 'first-class citizens' in a stronger sense that in the case

of 'classical' FP.

Expressions are of the form $e ::= X \mid c \mid f \mid (e_1, \ldots, e_n) \mid (e_1\ e_2)$, where $c \in DC$, $f \in FS$, and e_i are also expressions. As usual, application associates to the left and parentheses can be omitted accordingly. Therefore, $e\, e_1 \ldots e_n$ is the same as $(\ldots(\,(e\, e_1)\ e_2\,)\ldots)\, e_n)$. Of course, expressions are assumed to be well-typed. *First order patterns* are a special kind of expressions which can be understood as denoting data values, i.e., values not subject to further evaluation, in contrast to expressions, which can be possibly reduced by means of the rules of the program.

Each function $f \in FS^n$ is defined by a set of conditional rules $f\ t_1 \ldots t_n = e \Longleftarrow l_1 == r_1, \ldots, l_k == r_k$ where $(t_1 \ldots t_n)$ form a tuple of linear (i.e., with no repeated variable) patterns, and e, l_i, r_i are expressions. No other conditions (except well-typedness) are imposed to function definitions. Rules have a conditional reading: $f\ t_1 \ldots t_n$ can be reduced to e if all the equality conditions $l_1 == r_1, \ldots, l_k == r_k$ are satisfied (the condition part is omitted if $k = 0$). A predefined set of basic functions (as `map`, `foldl`, `take`, `repeat`, `zip`, etc.) is included, all of them with an equivalent semantics as in standard functional programming.

A distinguished feature of $\mathcal{TOY}(\mathcal{FD})$ is that no confluence properties are required for the programs, and therefore functions can be *non-deterministic*, i.e., return several values for a given (even ground) arguments. For example, the rules `coin = 0` and `coin = 1` constitute a valid definition for the 0-ary non-deterministic function `coin`. Two reductions are allowed, which lead to the values 0 and 1. The system tries in the first place the first rule, but, if backtracking is required by a later failure or by request of the user, the second rule is tried. Another way of introducing non-determinism in the definition of a function is by adding *extra* variables in the right side of the rules, as in `z_list = [0|L]` (where the usual list constructors [] and (:) are used). Any list of integers starting by 0 is a possible value of `z_list`. Anyway, note that in this case only one reduction is possible.

The language provides the polymorphic equality and disequality constraints == and /=, where `A == B` (respectively `A /= B`) succeeds if A and B can be reduced to the same first order pattern (respectively to different first order patterns). The repertoire of $\mathcal{FD}$ constraints is presented in Table 2.1. The relational constraints support reification [132]. Once again, == and /= are truly polymorphic, i.e., with the same operators for both $\mathcal{H}$ and $\mathcal{FD}$ solvers. Thus, $\mathcal{FD}$ variables are implicitly attributed to identify whether these equality and disequality constraints must be managed by either the $\mathcal{H}$ or the $\mathcal{FD}$ solver. The rest of relational constraints #>, #>=, #< and #<= post inequality constraints to the $\mathcal{FD}$ solver. Their arguments can be built with constraint operators, as #+, #-, #* and #/. To evaluate constraint expressions, each arithmetic constraint operator generates a new fresh variable Aux, which is constrained with the operation between the two constraint arguments. Thus, typical sum or product of a list of variables $[v_1, \ldots, v_n]$ is decomposed into $n - 1$ primitive constraints of two arguments, generating $n - 1$ auxiliary new variables.

Type	Constraints & Operators
Relational Constraints	==, /=, #>, #>=, #<, #<=
Arithmetic Operators	#+, #-, #*, #/
Propositional Constraint	post_implication
Domain Constraints	domain, domain_valArray
Global Constraints	all_different, count, sum, scalar_product

Table 2.1: Repertoire of $\mathcal{FD}$ Constraints and Operators

The propositional constraint `post_implication A B` posts to the solver a logical implication between the relational constraints A and B. The constraint `domain List Lb Ub` takes each variable X of `List`, and posts `X #>= Lb` and `X #<= Ub`. Differently, the constraint `domain_valArray Vars Vals` supports domain initialization of each variable of `Vars` to the set of values `Vals`.

The global constraint `sum List Op B` (where B can be an integer or an $\mathcal{FD}$ variable) improves the sum of n variables presented before, by saving the $n-1$ auxiliary new variables. Moreover, Op is a relational constraint symbol, and the relational constraint between B and the sum of the elements of `List` is also posted to the solver. Similarly, `scalar_product Vals Vars Op B` posts a relational constraint between B and the scalar product of the equally sized lists of variables and integer values `Vars` and `Vals`, respectively. `count A List Op B` imposes a relational constraint between B and the number of elements in `List` being equal to A. Finally, `all_different List` constrains all the variables of `List` to take different values.

2.4.2 FLP Program Example and Goals

This section presents a first $\mathcal{TOY}(\mathcal{FD})$ program in Figure 2.8, and several goals (i.e., goal-reduction procedures) in Figure 2.9. They give a brief overview of the pure FLP modeling features and goal solving mechanism of the system. CP($\mathcal{FD}$) issues are considered in next section.

Regarding the $\mathcal{TOY}(\mathcal{FD})$ program, it contains the operator (`//`), as well as the functions `repeat`, `take`, `gen_v_list`, `is_true` and `loop`:

- The operator (`//`) includes a polymorphic principal type `A -> A -> A`, with its two arguments being of any type A (but both of the very same one). The operator is non-deterministic, as its definition includes two rules with the same pattern matching for its two arguments. The rules return as a result the first and the second argument, respectively.
- The function `repeat` contains just one rule, generating an infinite list of repetitions of its input argument.

```
(//):: A -> A -> A
F // S = F
F // S = S
%
repeat:: A -> [A]
repeat X = [X | repeat X]
%
take:: int -> [A] -> [A]
take 0 Xs = []
take N [] = [] <== N > 0
take N [X|Xs] = [X| take (N-1) Xs] <== N > 0
%
gen_v_list:: [A]
gen_v_list = [X | gen_v_list]
%
is_true:: bool -> int
is_true true = 1
%
loop:: A
loop = loop
```

Figure 2.8: A First $\mathcal{TOY}(\mathcal{FD})$ Program

- The function `take` includes conditional rules. It receives the integer `N` an the list `Xs`, generating as a result a new list with the first `N` elements of `Xs`. The condition `N > 0` in the second and third rules are just to distinguish them from the first rule. Otherwise, the arguments `0 []` will match (pattern matching and unification) with the first two rules, and the arguments `0 [A|L]` will match with the first and third rules (in both cases, turning `take` into a non-deterministic function).
- The function `gen_v_list` contains a single ground rule, which generates an infinite list of new fresh variables (by using the extra variable `X` on the right hand side of the rule).
- The function `is_true` contains a single rule, which is applicable just if its argument is `true`.
- The function `loop` contains a single ground rule, which iterates causing the non termination of the computation.

Regarding the goals solved in the $\mathcal{TOY}(\mathcal{FD})$ session, they show the system support for: Types, polymorphism, principal type, curried expressions, non-deterministic functions, backtracking, lazy evaluation, conditional rules and call-time choice.

- Goal 1 shows a very basic non-deterministic computation. It solves `3 // 5 == X`, obtaining a first solution in which `X` is unified to 3 by applying the first rule of

```
% Goal 1
TOY(FD)> 3 // 5 == X
      { X -> 3 }
sol.1, more solutions (y/n/d/a) [y]?
      { X -> 5 }
sol.2, more solutions (y/n/d/a) [y]?
      no

% Goal 2
TOY(FD)> 3 // is_true Y == X
      { X -> 3 }
sol.1, more solutions (y/n/d/a) [y]?
      { Y -> true,
        X -> 1 }
sol.2, more solutions (y/n/d/a) [y]?
      no

% Goal 3
TOY(FD)> (take 3) // (take 2) == P
      { P -> (take 3) }
sol.1, more solutions (y/n/d/a) [y]?
      { P -> (take 2) }
sol.2, more solutions (y/n/d/a) [y]?
      no

% Goal 4
TOY(FD)> (take 3) // is_true == P
TYPE ERROR: Wrong type calling function or constructor symbol (// (take 3)
is_true) in goal. Argument 2 has type bool -> int but [ _A ] -> [ _A ] was
expected. Variable type is assumed to go on the inference.

% Goal 5
TOY(FD)> L == take 2 (repeat (0 // 1))
      { L -> [ 0, 0 ] }
sol.1, more solutions (y/n/d/a) [y]?
      { L -> [ 1, 1 ] }
sol.2, more solutions (y/n/d/a) [y]?
      no

% Goal 6
TOY(FD)> L == take 2 (repeat (take 1 gen_v_list))
      { L -> [ [ _A ], [ _A ] ] }
sol.1, more solutions (y/n/d/a) [y]?
      no

% Goal 7
TOY(FD)> X /= true, is_true X == 1, loop
      no
```

Figure 2.9: Solving Different Goals

(`//`). Then, if the user requests another solution, backtracking is performed, to apply any pending rule of (`//`).

- Goal 2 shows that evaluation of functions in $\mathcal{TOY}(\mathcal{FD})$ is not strict. The second argument of `3 // is_true Y` is not a pattern, but a expression that can be reduced. However, the first rule of (`//`) does not demand its second argument, and thus the rule can be applied without evaluating it (obtaining a first solution to the goal).

  Then, if the user requests another solution, backtracking is performed, and the exploration of the second rule of (`//`) demands the evaluation of the expression `is_true Y`. Lazy narrowing makes pattern matching of it with the single rule of `is_true`, unifying Y to `true`. As it can be seen, in the second solution found, both X and Y are unified to 1 and `true`, respectively.

- Goal 3 shows that any pattern value can be an argument of (`//`). In this case, both (`take 3`) and (`take 2`) are patterns (they both instantiate the type A with (`[B] -> [B]`)), as they are partial applications of the function `take`.

- Goal 4 shows that $\mathcal{TOY}(\mathcal{FD})$ is a strong typed system, performing type checking both at compilation and run time. In this case, the type (`bool -> int`) of `is_true` is not compatible with (`[B] -> [B]`)).

- Goal 5 shows an example of lazy evaluation, as the infinite list generated by `repeat` is partially computed on demand. Moreover, $\mathcal{TOY}(\mathcal{FD})$ follows a call-time choice semantics, where different occurrences of an expression in the rule of a function are *shared*, avoiding its evaluation more than once. In this case, (`0 // 1`) is computed just once, and shared for all the elements generated by `repeat`. As it can be seen, due to call-time choice only the solutions [0,0] and [1,1] are computed (instead of the four solutions [0,0], [0,1], [1,0] and [1,1]).

- Goal 6 shows a similar situation to Goal 5. In this case, the variable list (`take 1 gen_v_list`) used as an argument is computed just once, and then shared at any point of the computation. The underline in _A means that it is an extra variable, not taking place in any of the constraints of the proposed goal.

- Finally, Goal 7 shows that lazy narrowing tackles the constraints arisen in a $\mathcal{TOY}(\mathcal{FD})$ goal in textual order. First, the constraint `X /= true` is managed. Then, the management of `is_bool X == 1` triggers the evaluation of the single rule of `is_true`. However, lazy narrowing fails trying to unify X to `true`, as X was constrained to be different to it before. Thus, the goal fails, instead of looping (as it never reaches the evaluation of `loop`).

2.4.3 CFLP($\mathcal{FD}$) Program Examples and Goals

This section presents $\mathcal{TOY}(\mathcal{FD})$ as a CFLP($\mathcal{FD}$) system, providing two introductory examples dealing with $\mathcal{FD}$ constraints. First, the $\mathcal{TOY}(\mathcal{FD})$ model of the Queens specification (cf. Section 2.1.1) is presented in Figure 2.10. Then, the $\mathcal{TOY}(\mathcal{FD})$ model of the Golomb specification (cf. Section 2.1.2) is presented in Figure 2.11. The models are described next, and to ease this task the figures artificially include the lines of code. The figures also include a $\mathcal{TOY}(\mathcal{FD})$ session solving the instances Queens-4 and Golomb-5, respectively. As it can be seen, the two feasible solutions of Queens-4 and the optimal solution of Golomb-5 (cf. Sections 2.1.1 and 2.1.2) are obtained by the $\mathcal{TOY}(\mathcal{FD})$ goal computations.

Both Queens and Golomb models include the files `cflpfd.toy` and `misc.toy` (lines 1 and 2, respectively). The former contains the $\mathcal{FD}$ constraints of Table 2.1. The latter contains a sort of prelude, including the $\mathcal{TOY}(\mathcal{FD})$ version of FP standard primitive functions. The functions `map`, `foldl`, `zipWith`, `scanl`, `iterate`, `head`, `last` and `take` are used, as well as the operators `(++)` and `/\`. All of them have the same semantics as in standard FP [96], and their implementation as $\mathcal{TOY}(\mathcal{FD})$ functions are shown in Figure 2.12.

The main function `queens` (Figure 2.10, lines 3-10) models the CSP. Its single rule receives the input parameter `N`, representing the amount of queens to be placed. It computes as a result the list `L -> `$[v_1, \ldots, v_N]$, where v_i represents the row taken by the queen placed in the *i*-th column. To compute each feasible solution, the rule is turned into a conditional one, with six conditions (lines 5-10) to be hold:

- Line 5 generates a list of `N` new fresh logic variables, by using the extra function `gen_v_list` (lines 13-14).
- Line 6 constrains the domain of these variables to be in the range `1..N`.
- Line 7 constrains the variables to take different values, thus ensuring that no two queens are placed in the same row.
- Lines 8 and 9 constrain the variables to be placed in different decreasing and increasing diagonals, respectively. To do so, the lists $[v_1 + 0, \ldots, v_N + (N-1)]$ and $[v_1 - 0, \ldots, v_N - (N-1)]$ are constructed and constrained to take different values. In particular, the extra function `from` is used (lines 11-12), which receives the input argument `N` and computes the infinite list $[N, N+1, N+2, \ldots]$.
- Finally, line 10 uses the $\mathcal{FD}$ primitive `labeling` to specify a search strategy. The second argument specifies `L` as the variable list to be labeled. The first argument specifies a first fail variable selection criterion (i.e., select first the variable with smallest amount of domain values), using textual order for ties. As no value selection criterion is specified, the domain of each variable is labeled in an increasing order.

```
%
%---------------------------------
% INCLUDES
%---------------------------------
%
(01) include "cflpfd.toy"
(02) include "misc.toy"
%
%---------------------------------
% MAIN FUNCTION
%---------------------------------
%
(03) queens :: int -> [int]
(04) queens N = L <==
(05)    take N gen_v_list == L,
(06)    domain L 1 N,
(07)    all_different L,
(08)    all_different (zipWith (#+) L (take N (from 0))),
(09)    all_different (zipWith (#-) L (take N (from 0))),
(10)    labeling [ff] L
%
%---------------------------------
% EXTRA FUNCTIONS
%---------------------------------
%
(11) from:: int -> [int]
(12) from N = N : from (N+1)
%
(13) gen_v_list:: [A]
(14) gen_v_list = [X | gen_v_list]
%
%---------------------------------
% GOAL EXAMPLE
%---------------------------------
%
TOY(FD)> queens 4 == L
      { L -> [ 2, 4, 1, 3 ] }
sol.1, more solutions (y/n/d/a) [y]?
      { L -> [ 3, 1, 4, 2 ] }
sol.2, more solutions (y/n/d/a) [y]?
      no
```

Figure 2.10: Queens Model and System Session

```
%
%---------------------------------
% INCLUDES
%---------------------------------
%
(01) include "cflpfd.toy"
(02) include "misc.toy"
%
%---------------------------------
% MAIN FUNCTION
%---------------------------------
%
(03) golomb :: int -> [int]
(04) golomb N = M <==
(05)    M == take N [0 | gen_v_list],
(06)    order (M ++ [trunc(2^(N-1))]) == true,
(07)    gen_difs M == Ds,
(08)    foldl (++) [] Ds == D,
(09)    all_different D,
(10)    lbound Ds sums_nats == true,
(11)    (head D) #< (last D),
(12)    labeling [toMinimize (last M)] M
%
%---------------------------------
% EXTRA FUNCTIONS
%---------------------------------
%
(13) order:: [int]  -> bool
(14) order [X] = true
(15) order [X,Y|Xs] = (X #< Y) /\ order [Y|Xs]
%
(16) gen_difs:: [int] -> [[int]]
(17) gen_difs [] = []
(18) gen_difs [X|Xs] = [map (#- X) Xs|gen_difs Xs]
%
(19) lbound:: [[int]] -> [int] -> bool
(20) lbound Xss Is =
                foldl (/\) true (foldl (++) [] (map (zipWith (#<=) Is) Xss))
%
(21) sums_nats:: [int]
(22) sums_nats = scanl (+) 1 (iterate (+1) 2)
%
(23) gen_v_list:: [A]
(24) gen_v_list = [X | gen_v_list]
%
%---------------------------------
% GOAL EXAMPLE
%---------------------------------
%
TOY(FD)> golomb 5 == L
      { L -> [ 0, 1, 4, 9, 11 ] }
sol.1, more solutions (y/n/d/a) [y]?
      no
```

Figure 2.11: Golomb Model and System Session

```
map:: (A -> B) -> [A] -> [B]
map F [] = []
map F [X|Xs] = [(F X)|(map F Xs)]
%
foldl:: (A -> B -> A) -> A -> [B] -> A
foldl F Z [] = Z
foldl F Z [X|Xs] = foldl F (F Z X) Xs
%
zipWith:: (A -> B -> C) -> [A] -> [B] -> [C]
zipWith Z [] Bs = []
zipWith Z [A|As] [] = []
zipWith Z [A|As] [B|Bs] = [Z A B| zipWith Z As Bs]
%
scanl:: (A -> B -> A) -> A -> [B] -> [A]
scanl F Q [] = [Q]
scanl F Q [X|Xs] = [Q|scanl F (F Q X) Xs]
%
iterate :: (A -> A) -> A -> [A]
iterate F X = [X|iterate F (F X)]
%
head:: [A] -> A
head [X|_] = X
%
last:: [A] -> A
last [X] = X
last [_, Y|Xs] = last [Y|Xs]
%
take:: int -> [A] -> [A]
take 0 Xs = []
take N [] = [] <== N > 0
take N [X|Xs] = [X| take (N-1) Xs] <== N > 0
%
(++) :: [A] -> [A] -> [A]
[] ++ Ys = Ys
[X|Xs] ++ Ys = [X|Xs ++ Ys]
%
(/\):: bool -> bool -> bool
false /\ _ = false
true /\ X = X
```

Figure 2.12: Fragment of the File `misc.toy`

The main function `golomb` (Figure 2.11, lines 3-12) models the COP. Its single rule receives the input parameter `N`, representing the amount of marks to be placed in the ruler. It computes as a result the list `M -> [`$M_0, \ldots, M_{N-1}$`]`, where M_i represents the i-th mark of the ruler. To compute the optimal solution, the rule is turned into a conditional one, with eight conditions (lines 5-12) to be hold:

- Line 5 generates the list `M`, composed of an initial 0 (ensuring the problem requirement of M_0 = 0) and `N-1` new fresh logic variables, by using the extra function `gen_v_list` (lines 23-24).
- Line 6 uses the extra function `order` (lines 13-15) to constrain each element of `M` to be smaller than the following one. The function also sets a lower bound to the variables of `M`, as the first element of `M` is already 0. Moreover, by calling to `order` with `M ++ [trunc(2^(N-1))]` (where `++` represents the concatenation operator of lists), it also sets an upper bound for the mark variables.
- Line 7 uses the extra function `gen_difs` (lines 16-18) to generate the list of variables representing the differences between each pair of `M` variables. It computes the `[[int]] Ds =` $[[D_{1-0}, \ldots, D_{(N-1)-0}], [D_{2-1}, \ldots, D_{(N-1)-1}], \ldots, [D_{(N-1)-(N-2)}]]$. Each D_{i-j} is implicitly generated as the `Aux` variable of the arithmetic constraint $M_j - M_i$ (cf. the definition of the constraint operator `#-` in Section 2.4.1). As both M_i and M_j already have an associated domain, `gen_difs` also serves for domain initialization of the new generated variables.
- Line 8 converts `Ds` into the plain `[int] D =` $[D_{1-0}, \ldots, D_{(N-1)-0}, D_{2-1}, \ldots, D_{(N-1)-1}, \ldots, D_{(N-1)-(N-2)}]$.
- Line 9 constrains the variables of `D` to take different values.
- Line 10 uses the extra function `lbound` (lines 19-20) to constrain the lower bound of each D_{i-j} (considering that, as all the distances are pairwise distinct, their sum must be, at least, the sum of the first $j - i$ integers). The second argument of `lbound` uses another extra function `sums_nats` (lines 21-22). It computes the infinite list $[s_1, s_2 \ldots]$, where each s_i represents the sum of the first i natural numbers.

  The single rule of `lbound` applies `zipWith (#<=)` $[s_1, s_2 \ldots]$ to each element of `Ds`, lazily computing $[s_1, s_2 \ldots]$ and sharing it for the different elements of `Ds`. Taking as an example the first element of `Ds`, the computation of `zipWith (#<=)` $[s_1, s_2, s_{n-1}]$ $[D_{1-0}, \ldots, D_{(N-1)-0}]$ constrains $D_{1-0} >= s_1, \ldots, D_{(N-1)-0} >= s_{n-1}$. The higher-order application of `map` ensures that `zipWith` is applied to each element of `Ds`. As relational constraints support reification, assigning the result to no variable is a syntactic sugar for stating that just the `true` result is expected. Thus, the resulting list of applying `(map (zipWith (#<=) sum_nats) Ds)` is `[[bool]]` (with all the elements of the list being `true`). Then, `foldl (++) []`

converts `[[bool]]` into a plain `[bool]`. Finally, `foldl (/\) true` takes the `[bool]` list, returning `true` as a result (thus holding the equality constraint of line 10).

- Line 11 constrains the first variable of `D` to be smaller than the last one.
- Finally, line 12 uses the $\mathcal{FD}$ primitive `labeling` to specify a search strategy. The second argument specifies `M` as the variable list to be labeled. The first argument specifies a cost function for the search, stating that the optimal solution will be the one minimizing the last variable of `M`. As neither variable nor value selection criterion are specified, the variables are labeled in textual order, and the domain of each variable is labeled in an increasing order.

This finishes the preliminary concepts for the three research parts being accomplished in the thesis. Next parts present each of them.

Part II

Improving the Performance of $\mathcal{TOY}(\mathcal{FD})$

The first research part of the thesis is focused in improving the solving performance of $\mathcal{TOY}(\mathcal{FD})$. Chapter 3 describes the integration of external C++ CP($\mathcal{FD}$) solvers, with state-of-the-art performance, into $\mathcal{TOY}(\mathcal{FD})$. Chapter 4 describes the development of new search primitives, implemented in the new $\mathcal{TOY}(\mathcal{FD})$ versions interfacing C++ CP($\mathcal{FD}$) solvers.

Chapter 3

Interfacing External C++ CP($\mathcal{FD}$) Solvers

The CFLP($\mathcal{FD}$) system $\mathcal{TOY}(\mathcal{FD})$ provides a very suitable approach for modeling CSP's and COP's, as constraints are integrated into a declarative general-purpose language as FLP. First, the language provided is quite expressive: The logic component includes features such as relational notation, non-determinism, backtracking, logical variables, domain variables and the capability of reasoning with models. The functional component includes features such as functional notation, curried expressions, higher-order functions, patterns, partial applications, lazy evaluation, types and polymorphism. Second, constraint solving is simply added to the system by interfacing a constraint solver. The original $\mathcal{TOY}(\mathcal{FD})$ version this thesis is departing from [72] interfaces the native CP($\mathcal{FD}$) library SICStus `clpfd`. The use of an interface penalizes the performance of the system, which must now intermix constraint solving with lazy narrowing (and thus an inherent overhead comes from coordinating the solver).

The Amdahl's law establishes that the improvement in the performance of a system due to the alteration of one of its components is limited by the fraction of time the component is used. Due to the combinatorial nature of the CSP's and COP's tackled with $\mathcal{TOY}(\mathcal{FD})$ it is expected that, as long as the instances scale up enough, most of the CPU time the system requires to solve them is devoted to $\mathcal{FD}$ solver computations. Thus, a suitable approach to improve the solving efficiency of $\mathcal{TOY}(\mathcal{FD})$ is to develop new system versions which, instead of rely on the underlying SICStus `clpfd` solver, rely on other external solvers with state-of-the-art performance. That is, stick to the same $\mathcal{TOY}(\mathcal{FD})$ model of the problems (same $\mathcal{FD}$ constraint network and search strategy) and rely on new constraint solvers capable of dealing with these computations faster.

The main contribution of this chapter is to present the development of two new $\mathcal{TOY}(\mathcal{FD})$ versions: $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$, relying on the external solvers of Gecode and ILOG Solver (respectively). Also, the functionality of the original $\mathcal{TOY}(\mathcal{FD})$

version (referred to as $\mathcal{TOY}(\mathcal{FD}s)$ from now on, as it relies on SICStus `clpfd`) is enhanced with new propagation mode capabilities also implemented for $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$. More generally, the chapter describes a generic scheme for interfacing C++ CP($\mathcal{FD}$) solvers to $\mathcal{TOY}(\mathcal{FD})$, in a setting easily adaptable to any other CLP($\mathcal{FD}$) or CFLP($\mathcal{FD}$) system implemented in Prolog. Section 3.1 describes the architecture of the system, focusing on the interface to its $\mathcal{FD}$ solver and identifying the requirements to coordinate it. Section 3.2 describes the scheme for interfacing C++ CP($\mathcal{FD}$) solvers, and Section 3.3 instantiates it with Gecode and ILOG Solver. Section 3.4 analyzes the new $\mathcal{TOY}(\mathcal{FD})$ performance achieved by using a set of benchmark problems. Section 3.5 presents some related work and, finally, Section 3.6 reports conclusions.

3.1 $\mathcal{TOY}(\mathcal{FD})$ Architecture

$\mathcal{TOY}(\mathcal{FD})$ is implemented in SICStus Prolog [126] and, instead of using an abstract machine for running byte-code or intermediate code from compiled programs, the $\mathcal{TOY}$ compiler uses SICStus Prolog as an object language [122]. Its architecture contains a Herbrand ($\mathcal{H}$) constraint solver, dealing with syntactic equality/disequality constraints, and an $\mathcal{FD}$ constraint solver, dealing with Finite Domain constraints. More specifically, each solver can be characterized by a constraint store and a constraint engine: The store contains the set of constraints to be satisfied and the engine solves it.

Whereas the $\mathcal{FD}$ solver is interfaced to the system, the $\mathcal{H}$ solver is implemented on plain Prolog (easing its integration as an inherent component of the architecture). Only disequality constraints (`/=`) are maintained in an explicit store ($\texttt{store}^{\mathcal{H}}$), and thus its engine consists of explicitly controlling store consistency when processing each new goal constraint. Equality constraints (==) are processed via Prolog unification, taking into account the store of disequality constraints [72].

Figure 3.1 presents a $\mathcal{TOY}(\mathcal{FD})$ program (top) and goal (bottom), showing how lazy narrowing performs goal solving relying on the $\mathcal{H}$ and $\mathcal{FD}$ solvers. The function `bin` contains a single rule returning `true` by constraining the list of variables received as its argument to be binary. The non-deterministic and polymorphic operator `(///)` is defined by two rules, returning either its first or its second argument. Then, the $\mathcal{TOY}(\mathcal{FD})$ goal proposed consists of the conjunction of two syntactic equality constraints over expressions, for which three solutions are found.

The solving process followed by the system to find those solutions proceeds as follows: First, the constraint `bin [X] == true` is selected. Whereas its right argument `true` is a constant (pattern), its left argument `bin [X]` is an expression. Thus, lazy narrowing triggers the first rule of the function `bin`, performing the unification of `L1` and `[X]` (where `L1` is an instance of variable `L` in `bin` rule). It also posts the constraint `domain [X] 0 1` to the $\mathcal{FD}$ solver. The `bin` rule returns `true`, and the initial constraint becomes the primitive one `true == true`. It is then posted to the $\mathcal{H}$ solver,

```
include "cflpfd.toy"
%
bin:: [int] -> bool
bin L = true <==
  domain L 0 1
%
(//) :: A -> A -> A
F // S = F
F // S = S
-----------------------------------------------------
TOY(FD)> bin [X] == true, (X // Y) #> 5 == R
sol1: {R -> false} /\ {X in 0..1} ? ;
sol2: {R -> true} /\ {X in 0..1, Y in 6..sup} ? ;
sol3: {R -> false} /\ {X in 0..1, Y in -inf..5} ? ;
no
```

Figure 3.1: $\mathcal{TOY}(\mathcal{FD})$ Program and Goal Example

which straightforwardly success.

Then, the constraint `(X // Y) #> 5 == R` is selected. Whereas R is a variable (pattern), `(X // Y) #> 5` is an expression, which needs its first argument to be reduced (in order to post the constraint to the $\mathcal{FD}$ solver). Thus, lazy narrowing triggers the first rule of `(//)`, using a new fresh variant of each variable of the rule (in this case, F1 for F and S1 for S), and performing the unification of F1 and X (respectively of S1 and Y). The rule returns X as a result. The constraint #> supports reification, and thus either `X #> 5` or `X #<= 5` can be applied (respectively returning `true` and `false`). The system selects first `X #> 5`, but then the $\mathcal{FD}$ solver fails to find a solution, as X was previously constrained to be binary. Thus, the system backtracks to #> (removing the constraint `X #> 5` from the $\mathcal{FD}$ solver) and selects `X #<= 5`, returning `false`. Constraint propagation succeeds, and computation continues. The initial constraint becomes the primitive one `false == R`, which is posted to the $\mathcal{H}$ solver, unifying R.

A first solution has been obtained, consisting in the non-ground constraints of the $\mathcal{H}$ and $\mathcal{FD}$ stores. If a new solution is requested, the system backtracks to the application of `(X // Y)`, and both $\mathcal{H}$ and $\mathcal{FD}$ stores are restored to the contents they had before `(X // Y) #> 5` was processed. The unexplored second rule of `(//)` is then applied, continuing the computation in a similar way.

3.1.1 Coordinating the CP($\mathcal{FD}$) Solver

The $\mathcal{FD}$ solver must be interfaced to the system, coordinating its execution within the lazy narrowing operational semantics. Figure 3.2 presents the interface allowing $\mathcal{TOY}(\mathcal{FD})$ to interact with $\texttt{engine}^{\mathcal{FD}}$ by sending the following commands to it:

- Posting a new constraint C to $\texttt{store}^{\mathcal{FD}}$. If the constraint is not a primitive one,

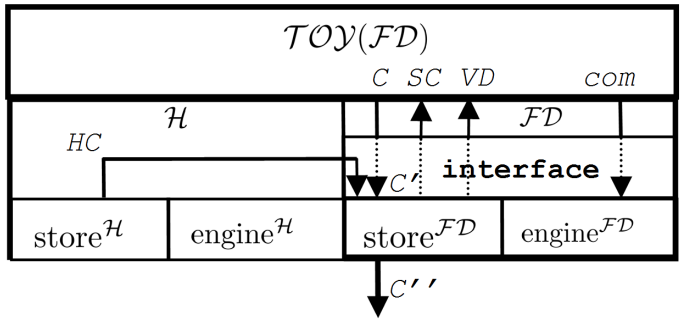

Figure 3.2: $\mathcal{TOY}(\mathcal{FD})$ Interface to its $\mathcal{FD}$ Solver

its arguments must be previously narrowed (transforming C into one or more primitive constraints C'). For example, the constraint $C \equiv$ `(#>5) (X // Y)` must narrow its second argument, sending the constraint $C' \equiv$ `X #> 5` to the solver. Also, the constraint $C \equiv$ `domain [X #+ Y, X #- Y] 0 1` must narrow its first argument, sending to the solver $C' \equiv$ {`X#+Y == _A, X#-Y == _B, domain[_A, _B] 0 1`}.

- Due to the polymorphism of `==` and `/=`, $\mathcal{FD}$ variables are attributed [105] to identify whether these equality and disequality constraints must be managed by the $\mathcal{FD}$ solver or by the $\mathcal{H}$ one.

  That it, the constraints of a goal are processed in their textual order. Each time lazy narrowing processes a `==` constraint: It is managed by `engine`$^{\mathcal{FD}}$ iff both of its arguments are $\mathcal{FD}$ variables; Otherwise, it is managed by `engine`$^{\mathcal{H}}$. Similarly, each time lazy narrowing processes a `/=` constraint: It is managed by `engine`$^{\mathcal{FD}}$ iff both of its arguments are $\mathcal{FD}$ variables, or one of its arguments is an integer and the other one is a non $\mathcal{FD}$ attributed variable; Otherwise, it is managed by `engine`$^{\mathcal{H}}$.

  However, there is one more issue as, due to order in which the constraints of a goal are processed, it is possible that disequality constraints initially sent to `store`$^{\mathcal{H}}$ must be further transferred to `store`$^{\mathcal{FD}}$. For example, Figure 3.3 presents a goal in which the first four constraints are posted to the $\mathcal{H}$ solver (in particular, unifying Y with 1). However, when processing `X #> 0`, the system attributes X as an $\mathcal{FD}$ variable, and the constraints of `store`$^{\mathcal{H}}$ `X /= 1` and `X /= Z` become relevant for the $\mathcal{FD}$ solver, so they must be transferred to `store`$^{\mathcal{FD}}$ (attributing also Z as an $\mathcal{FD}$ variable, which recursively transfers `1 /= Z` to `store`$^{\mathcal{FD}}$).

- Removing remaining constraints C'' of `store`$^{\mathcal{FD}}$, keeping it consistent with the computational point the system has backtracked to.

- Performing constraint propagation over `store`$^{\mathcal{FD}}$, simplifying it or detecting fail-

```
TOY(FD)> X /= Y, X /= Z, Y /= Z, Y == 1, X #> 0
sol1: {Y->1} /\ {Z /= X, X in 2..sup, Z in(inf..0)\/(2..sup)} ? ;
no
```

Figure 3.3: $\mathcal{TOY}(\mathcal{FD})$ Second Goal Example

ure in advance.

- Performing a search strategy over $\texttt{store}^{\mathcal{FD}}$, looking for a feasible/optimal new solution.
- Getting the non-ground constraints of $\texttt{store}^{\mathcal{FD}}$ *SC* and the domains of its associated $\mathcal{FD}$ variables *VD*, to show them in the solution.

3.1.2 Implementing the Interface

The interface access to the $\mathcal{FD}$ solver consists of a set of Prolog predicates (Figure 3.4 presents `interface` as the set of Prolog predicates $pred_1, \ldots, pred_k$, each of them implementing a system command by using the solver API) In $\mathcal{TOY}(\mathcal{FD}s)$, $\mathcal{TOY}$ and `clpfd` variables have a unique representation as Prolog logical variables. Also, `clpfd` provides an implicit management of backtracking (restoring $\texttt{store}^{\mathcal{FD}}$ to the computational point the system has backtracked to) and of multiple searches interleaved with constraint posting (each search acting only over its associated constraint network). Finally, `clpfd` implicitly performs incremental propagation mode (the engine performs constraint propagation each time a new constraint is posted to the store [197]).

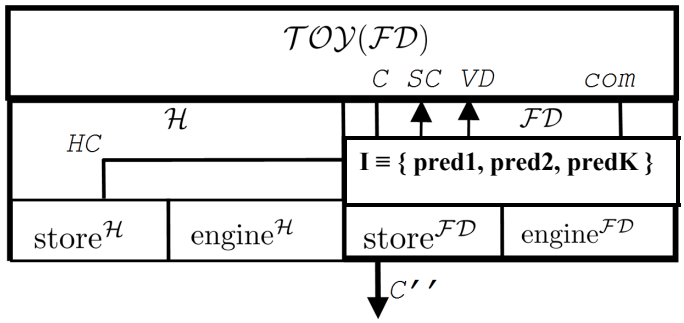

Figure 3.4: $\mathcal{TOY}(\mathcal{FD})$ Interface Set of Prolog Predicates

3.2 Generic Scheme for Interfacing External Solvers

An approach to improve the constraint solving performance of $\mathcal{TOY}(\mathcal{FD})$ consists of integrating new state-of-the-art C++ CP($\mathcal{FD}$) solvers, more efficient than `clpfd`. However, their integration into $\mathcal{TOY}(\mathcal{FD})$ increases the complexity of `interface`. The first difficulty comes from the communication between system and solver (which now requires connecting components implemented in different languages). Also, variables, constraints and types differ between systems (impedance mismatch). Three additional difficulties come from adapting a C++ CP($\mathcal{FD}$) solver to the requirements of a CFLP($\mathcal{FD}$) system. First, *model reasoning* (associated to the non-determinism of logic programming) is not a target in the constraint language. Thus, the solver API does not provide a method for removing a single constraint from the store. Second, a single search strategy is expected to be applied to the whole constraint network. To use multiple searches interleaved with constraint posting it is necessary to generate several solvers (`storeFD` and `engineFD`). Third, incremental propagation [197] may not be supported, as a first constraint propagation is expected to be performed over the whole constraint network just before the search starts. The following subsections present how to overcome such difficulties.

3.2.1 Communication

The Prolog predicates $pred_1, \ldots, pred_k$ acting as an interface to the solver API require now communication with C++ code. SICStus provides a Prolog-C++ communication framework allowing to define a Prolog predicate prototype whose implementation is contained in a C++ function. The prototype includes the number of arguments, specifying its mode (input/output) and type. There is a conversion between Prolog and C++ parameters, including a C++ representation of Prolog terms.

Figure 3.5 presents the modified `interface`, where the functionality of the $\mathcal{TOY}(\mathcal{FD})$ commands is implemented by a set of C++ functions $f_1, \ldots, f_n$, accessing to the solver API. For example, f_i can create a new $\mathcal{FD}$ variable, f_j can post a `domain` constraint and f_z can perform constraint propagation. These C++ functions are then made accessible to the Prolog code by their mapping to the Prolog prototype predicates $p_1, \ldots, p_n$. Finally, any predicate $pred_i$ of `interface` can use p_i, p_j and p_z as many times as needed to coordinate `engine`$^{\mathcal{FD}}$.

Orthogonally, the interface must be extended with Prolog and C++ data structures, visible to the Prolog predicates and acting as a glue for solver integration. Whereas C++ data structures are stored as global variables, Prolog ones are stored in the term `fd_glue`. `store`$^{\mathcal{H}}$ is modified to be the pair (`store`$^{\mathcal{H}}$, `fd_glue`), where its second component represents the current Prolog data structures.

The rest of the section presents the different issues being tackled, analyzing for each of them the interaction with these data structures.

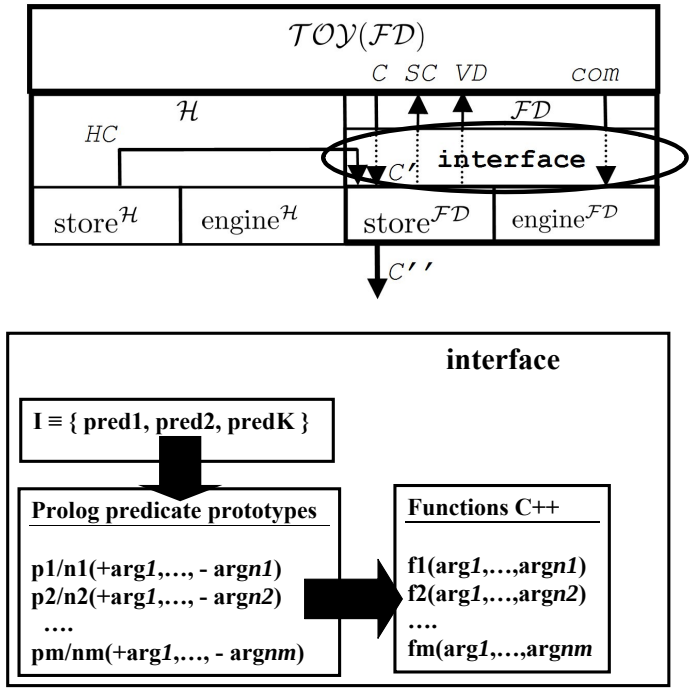

Figure 3.5: Map of Prolog Predicates to C++ Functions

3.2.2 Representation

Each C++ CP($\mathcal{FD}$) solver provides its own constraint and decision variable representations, different from those of $\mathcal{TOY}(\mathcal{FD})$. Thus, at implementation level, each variable and constraint contain two different but mate representations, and the interface must supply such representation mapping. For each new system constraint pC_i arisen in the goal (see Figure 3.6), the interface generates a mate solver constraint cC_i, posting it to store$^{\mathcal{FD}}$. Also, $\mathcal{FD}$ variables are explicitly attributed on each $\mathcal{FD}$ constraint arisen to further detect equality or disequality constraints belonging to $\mathcal{FD}$.

Variable communication is bidirectional. On the one hand, when processing pC_i, its logic variables $\mathsf{pV}_1 \ldots \mathsf{pV}_k$ must be mapped to mate decision variables $\mathsf{cV}_1 \ldots \mathsf{cV}_k$, posting cC_i over them. However, both the Prolog engine and store$^{\mathcal{FD}}$ lack API methods for obtaining their stored variables. Thus, the vectors pV and cV are used, with pV_i and cV_i representing the mate logic and decision representations of the *i*-th variable arisen in the goal computation. On the other hand, variable domains and non-ground constraints of the stores are shown in $\mathcal{TOY}(\mathcal{FD})$ solutions (cf. Section 3.1), using the information of pV and pC (respectively). First, it is thus necessary that cV variables bound by constraint propagation trigger the unification of their mate pV variables. To do so, the unary daemon constraint class d is defined, which posts a constraint to store$^{\mathcal{FD}}$ on each decision variable cV_i created, and propagates when the variable is bound to

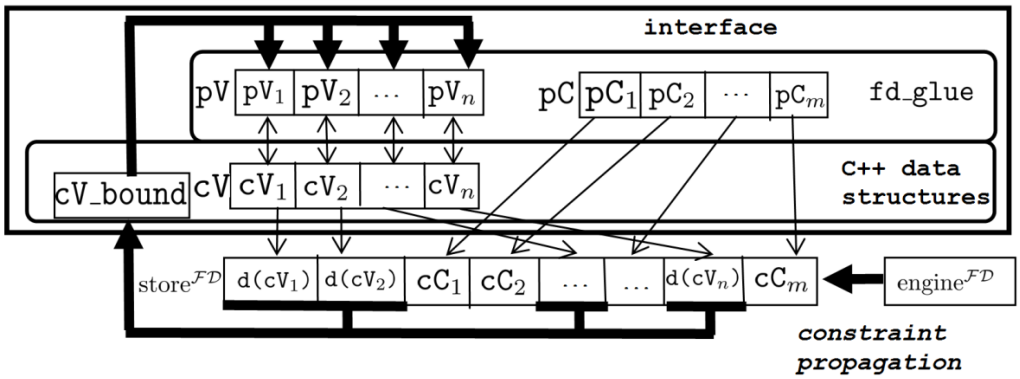

Figure 3.6: Data Structures

value k, storing the pair `(i, k)` into the C++ data structure cV_bound. It is emptied before constraint propagation, and traversed then to collect cV variables which had become bound, generating a Prolog term used by the Prolog interface predicates to unify mate pV variables. Second, given a variable as argument, the solver API provides a method for returning its domain. Thus, cV is traversed, requesting the domain of each unbound $\mathtt{cV}_i$. Finally, whereas `clpfd` provides a method for retrieving the whole content of $\mathtt{store}^{\mathcal{FD}}$, a C++ solver does not. Thus, each processed $\mathtt{pC}_i$ to be made explicit in the Prolog vector pC, to show them as part of $\mathcal{TOY}(\mathcal{FD})$ solutions. Regarding pV and cV consistency, pC is traversed, showing only any non-ground constraints.

3.2.3 Backtracking

Model reasoning offers appealing possibilities for constraint model design (see [134] for an example where chronological backtracking plays a key role on improving the performance of solving an academic timetabling problem). It is supported in $\mathcal{TOY}(\mathcal{FD})$ via non-deterministic functions, whose multiple rules are explored by backtracking, restoring $\mathtt{store}^{\mathcal{H}}$ and $\mathtt{store}^{\mathcal{FD}}$. The $\mathcal{TOY}(\mathcal{FD})$ operational mechanism processes the constraints of a goal in order, from left to right. Thus, backtracking consists of removing the k processed primitive constraints from $\mathtt{store}^{\mathcal{H}}$ and $\mathtt{store}^{\mathcal{FD}}$ since the last choice point. For example, in the goal `bin [X] == true, (#>5) (X // Y) == R` backtracking removes {`F1 == X, S1 == Y, R == false`} from $\mathcal{H}$ and {`X #<= 5`} from $\mathcal{FD}$ before starting to explore the second rule of `//` (cf. Section 3.1). A backtracking affects the $\mathcal{FD}$ solver if the last $\mathtt{k1} \in \{1, \ldots, \mathtt{k}\}$ $\mathcal{FD}$ primitive constraints are removed, also involving the last $\mathtt{k2} \geq 0$ $\mathcal{FD}$ variables created (associated only to those k1 constraints).

$\mathtt{store}^{\mathcal{H}}$ and `fd_glue` are automatically restored by the Prolog engine, but both $\mathtt{store}^{\mathcal{FD}}$ and C++ data structures are not. Figure 3.7 presents an example where pV and pC elements which are automatically removed are shown in dashed lines. Only Prolog interface predicates identify a backtracking affecting $\mathcal{FD}$ solver by simply com-

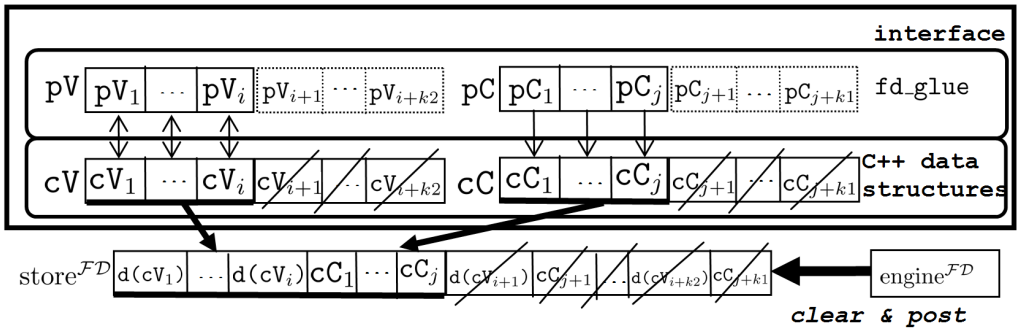

Figure 3.7: Backtracking Identification and Restoration

paring `store`$^{\mathcal{FD}}$ and pV+pC sizes. In this setting, some new $\mathcal{H}$ constraints can be processed with an unsynchronized $\mathcal{FD}$ solver, but as soon as any interaction with the solver is requested, the associated Prolog predicate will detect the backtracking and trigger the necessary command to synchronize the solver.

To remove the remaining k2 variables of cV, the sizes of pV and cV must be first compared. To remove remaining k1+k2 constraints of `store`$^{\mathcal{FD}}$, it is first cleared and then `i+j` constraints are posted again (rebuilding the whole model, as the solver API does not provide a method for removing a single constraint from `store`$^{\mathcal{FD}}$). Whereas cV is traversed to create and post again the unary daemon d constraints (cf. Section 3.2.2), for posting the primitive constraints two possibilities are considered. The first one traverses pC, translating again from each pC_i to its mate cC_i. The second one replicates the primitive constraints posted to `store` in the vector cC. This second option is selected, as in general managing cC is faster than translating pC_i constraints. To remove the remaining k1 constraints of cC, the sizes of pC and cC must be first compared. Finally, cC is traversed to post again the constraints over `store`$^{\mathcal{FD}}$.

3.2.4 Search Strategies

Multiple labelings interleaved with posting of constraints offer new possibilities for constraint model design (see [207] for an academic timetabling of two consecutive stages, each of them with a set of constraints and a labeling). In $\mathcal{TOY}(\mathcal{FD})$, this feature is supported (a `labeling` is just an expression), but in most C++ CP($\mathcal{FD}$) solvers the API method for posting new constraints to `store`$^{\mathcal{FD}}$ is precluded while `engine`$^{\mathcal{FD}}$ executes the search. For example, Figure 3.8 presents a goal in which, after posting `domain [X,Y] 0 2`, `labeling [] [X]` triggers the search mode of `engine`$^{\mathcal{FD}}$. The search finds a first solution (`X -> 0`), but the goal solving process must continue (processing `Y #> X`, `labeling [] [Y]`) with `engine`$^{\mathcal{FD}}$ in active search mode, as there is still remaining search space of `labeling [] [X]` (which, in fact, contains another feasible solution, with `X -> 1`).

```
TOY(FD)> domain [X,Y] 0 2, labeling [] [X], Y#>X, labeling [] [Y]
sol1: {X -> 0, Y -> 1} /\ {} ? ;
sol2: {X -> 0, Y -> 2} /\ {} ? ;
sol3: {X -> 1, Y -> 2} /\ {} ? ;
no
```

Figure 3.8: $\mathcal{TOY}(\mathcal{FD})$ Third Goal Example

Thus, `interface` must be adapted to handle this situation, further supporting the posting of `Y #> X` to `store`$^{\mathcal{FD}}$ and the new `labeling [] [Y]`. The search exploration of the latter leads to the first two solutions displayed. Then, the system backtracks to `labeling [] [X]`, finding a new value for X and continues the computation by posting again `Y #> X` and `labeling [] [Y]`, which finds the third solution.

To support these scenarios, `store`$^{\mathcal{FD}}$ is devoted to constraint posting and a vector of auxiliary solvers `ss` is used for managing the labelings $\mathtt{l}_1 \ \ldots \ \mathtt{l}_l$ arisen along a goal computation (see Figure 3.9). Thus, `engine`$^{\mathcal{FD}}$ never executes a labeling (and thus never reaches an active search mode), and $\mathtt{ss}_i$ just cares about $\mathtt{l}_i$, not managing any of the regular constraints arisen in the goal after that particular `labeling` (they will be managed by `engine`$^{\mathcal{FD}}$).

In this setting, multiple labelings can be performed interleaved with constraint posting by simply synchronizing `store`$^{\mathcal{FD}}$ with each $\mathtt{store\_ss}_i$:

- When creating a new solver $\mathtt{ss}_i$, cC is traversed, posting its constraints to $\mathtt{store\_ss}_i$. No daemon d constraints are needed as, during search, those paths leading to no solutions will bind variables to wrong values. Thus, $\mathtt{store\_ss}_i$ is consistent with the current state of goal computation and $\mathtt{l}_i$ is passed to $\mathtt{engine\_ss}_i$, which looks for its solutions.
- As a labeling is an enumeration process, the path leading $\mathtt{engine\_ss}_i$ to a solution consists of a set of (variable, value) equality constraints, which are added to $\mathtt{store\_ss}_i$. This solution path (set of constraints) must be posted to cC and `store`$^{\mathcal{FD}}$, to make them consistent with the effect of performing the search, and then continue goal computation. To do so, cV is traversed, requesting $\mathtt{engine\_ss}_i$ for each $\mathtt{cV}_i$ domain, and posting the equality constraint if it is bound to a value.

Regarding `interface`, the Prolog predicate devoted to manage labeling expressions takes $\mathtt{l}_i$ as argument, and uses a `repeat` loop to request $\mathtt{engine\_ss}_i$ to look for its solutions one by one. To distinguish the first invocation to $\mathtt{l}_i$ (requesting the creation of $\mathtt{ss}_i$) from further ones (requesting just another solution by backtracking) the size of `ss` is replicated in `fd_glue`, being automatically restored on backtracking.

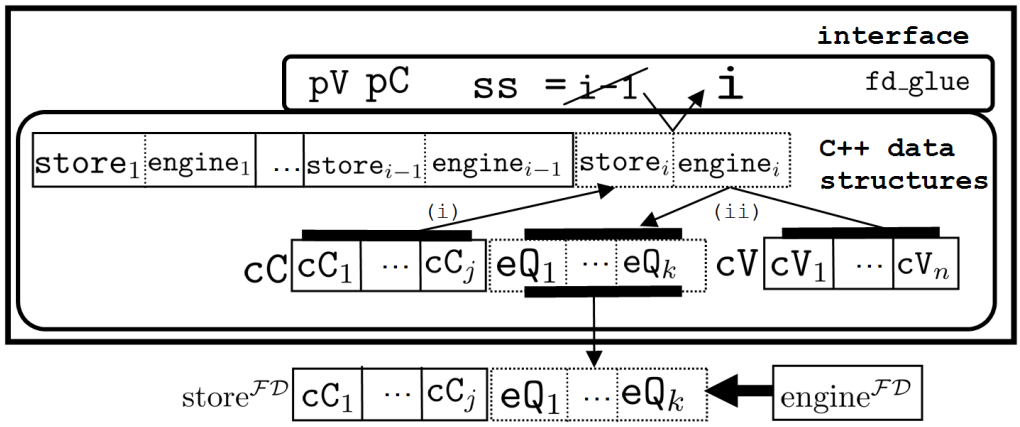

Figure 3.9: Labeling Management

On each loop iteration, the Prolog predicate compares both sizes. If no more solutions are found, $\texttt{ss}_i$ is deleted and the whole loop fails.

3.2.5 Incremental and Batch Propagation

Incremental propagation mode is usual in systems supporting reasoning with models, as they can detect failure along goal computations as soon as possible. However, batch propagation mode (the engine performs constraint propagation on demand) is more usual in C++ CP($\mathcal{FD}$) applications, and can enhance solving performance by saving constraint store propagations. However, it is possible that the evaluation of goals leading to no solutions continues until a propagation is demanded [197]. $\mathcal{TOY}(\mathcal{FD})$ primitives `batch_on` and `batch_off` are defined, enabling and disabling batch mode (respectively). The latter reestablishes incremental mode, triggering a $\texttt{store}^{\mathcal{FD}}$ constraint propagation before processing any new constraint. Both primitives can be freely used in $\mathcal{TOY}(\mathcal{FD})$ programs, and users can decide which parts of the goal are to be solved with incremental or batch modes.

When interfacing C++ CP($\mathcal{FD}$) solvers, batch mode implies that constraints are posted to `cC` but not to $\texttt{store}^{\mathcal{FD}}$. Also, no $\texttt{d(cV}_i\texttt{)}$ is created and posted for each new $\texttt{cV}_i$ created. When incremental mode is enabled, the remaining constraints must be posted to $\texttt{store}^{\mathcal{FD}}$ before performing constraint propagation. Thus, the tuple of integers (b, bcV, bcC) is added to `fd_glue`. Whereas b is a binary value representing the propagation mode, bcV and bcC represent the number of d and cC constraints posted to $\texttt{store}^{\mathcal{FD}}$ (respectively). First, when a `batch_on` primitive is processed, b is set to 1, and bcV/bcC are set to the current cV/cC sizes. Top of Figure 3.10 describes this situation. Second, when a `batch_off` primitive is processed, b is set to 0, and bcV/bcC are compared to cV/cC sizes, posting to $\texttt{store}^{\mathcal{FD}}$ the remaining constraints. Middle of Figure 3.10 describes this situation, where `cVj-1` and `cVj` are posted to

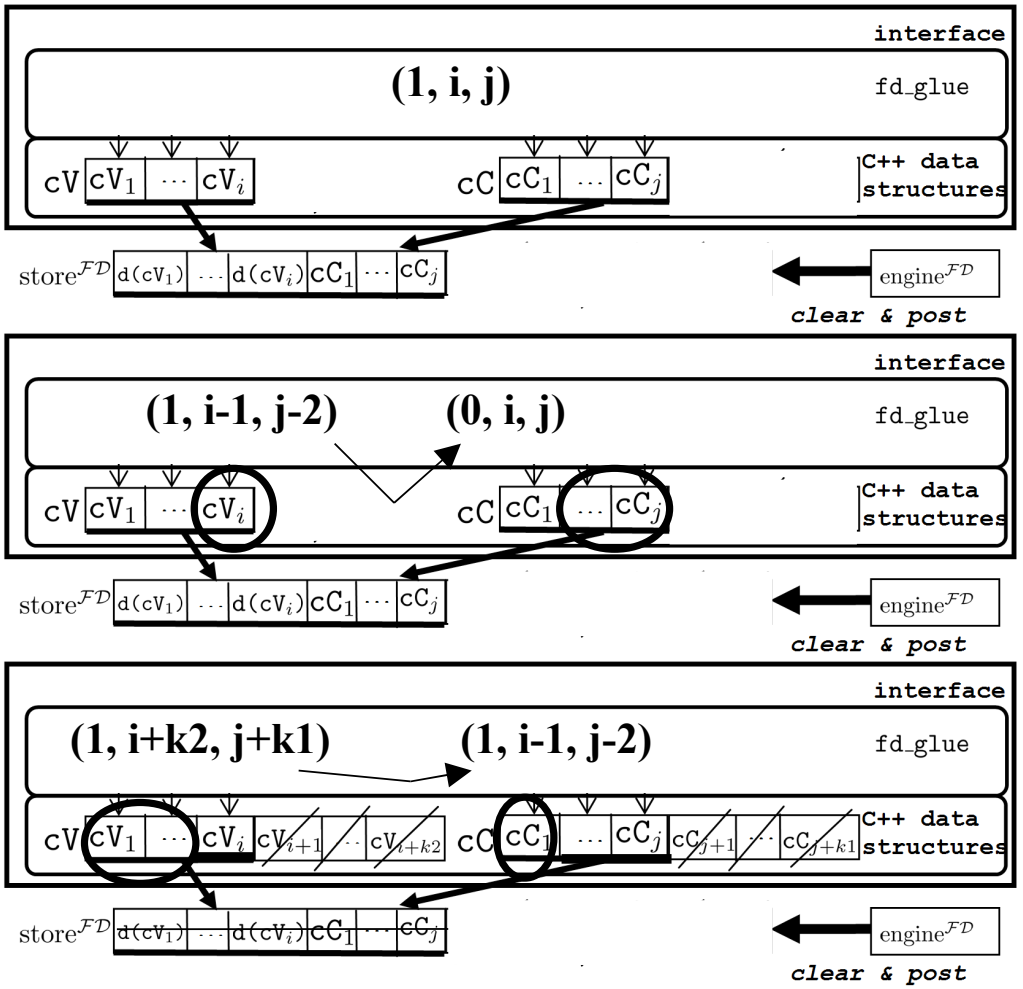

Figure 3.10: Batch and Incremental Propagation Modes

storeFD and cVi also triggers the posting of d(cVi). Third, when a backtracking is performed to a point where b is set to 1, restored sizes of bcV and bcC are used to post to $\texttt{store}^{\mathcal{FD}}$ only those constraints processed before last batch_on primitive was processed. Bottom of Figure 3.10 describes this situation, where backtracking to i-1 (respectively j-2) triggers the posting of d(cV1),...,d(cVi-1), but not of d(cVi) (respectively cC1,...,cCj-2, but not of cCj-1 and cCj).

Batch mode has been also implemented in $\mathcal{TOY}(\mathcal{FD}s)$, using SICStus delayed goals. $\texttt{store}^{\mathcal{H}}$ is modified to be the pair ($\texttt{store}^{\mathcal{H}}$, B), including B as the current propagation mode. When a batch_on primitive is processed, a new logic variable is placed on B, freezing the posting of each constraint on its grounding. When a batch_off primitive is processed, B is bound to a ground term, and frozen constraints are automatically posted to $\texttt{store}^{\mathcal{FD}}$.

3.3 Instantiating the Scheme

By interfacing two different C++ CP($\mathcal{FD}$) solvers to $\mathcal{TOY}(\mathcal{FD})$, the described scheme is shown to be generic enough. The solvers Gecode 3.7.3 [78] and IBM ILOG Solver 6.8 [12] are selected to develop the new versions $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ (respectively). Both these new versions and the original $\mathcal{TOY}(\mathcal{FD}s)$ one are available at: `http://gpd.sip.ucm.es/ncasti/TOY(FD).zip`.

Regarding $\mathcal{TOY}(\mathcal{FD}g)$, Gecode 3.7.3 has been selected as it is an open source constraint solver with state-of-the-art performance (awarded with the best results in the last five editions of the MiniZinc Challenge [7]). It provides a wide set of templates allowing users to be first class citizens in the development of new variable domains, constraints and search strategies. Regarding $\mathcal{TOY}(\mathcal{FD}i)$, ILOG Solver 6.8 has been selected as it is an industrial market leader for solving generic $\mathcal{FD}$ problems. It belongs to the ILOG CP 1.6 package, which contains the ILOG Concert 12.2 modeling library and two other solver libraries for specific scheduling and routing problems. Thanks to the IBM academic initiative these products are free for academic purposes.

3.3.1 Integration of Gecode: $\mathcal{TOY}(\mathcal{FD}g)$

Gecode provides a single library for solving CP($\mathcal{FD}$) problems. Its constraint catalogue includes integer, Boolean, set, scheduling and graph constraints. An object `Space` contains a store and an engine, thus being a solver. An application must define its own class inheriting from `Space`, and constraint network and search strategies are specified directly on it. First, a variable is explicitly represented as an `IntVar` object, and `IntVarArgs` and `IntVarArray` are provided as dynamic and static variable containers (respectively), to be part of the class attributes. Second, a constraint is implemented as a set of propagators applied to variables. There is not an explicit constraint object, but several API methods, each of them devoted to a different type of constraint, posting the suitable propagators implementing it. For optimization problems, the cost function must be associated to a variable, to apply branch and bound techniques [100] on it. Third, a `Branch` object represents a labeling to be applied to a `Space`. A `SearchEngine` object performs the search by first wrapping a `Space` containing at least one labeling. It uses hybrid recomputation techniques based on cloning `Spaces`, instead of trailing (see [157] for a comparison). Two `Spaces` are equivalent if they contain equivalent stores. A method for cloning an `IntVarArray` is provided (not for `IntVarArgs`), including the propagators posted on them. Thus, cloning the `IntVarArray` suffices to clone the whole `Space`. If search succeeds, the `SearchEngine` returns the `Space` obtained from applying the solution path.

The scheme of Section 3.2 is instantiated to develop $\mathcal{TOY}(\mathcal{FD}g)$, with Gecode 3.7.3 as its $\mathcal{FD}$ solver. First, a `Solver` class inheriting from `Space` is defined, representing $\texttt{store}^{\mathcal{FD}}$ and $\texttt{engine}^{\mathcal{FD}}$. As it must deal with any constraint network arisen in

a $\mathcal{TOY}(\mathcal{FD})$ goal, the number of needed variables is not known in advance. Thus, an `IntVarArgs` container is used as a `Solver` attribute to represent cV. Second, as there is no native constraint representation to be stored in cC, then a representation consisting of a vector of integers is defined, representing the kind of the constraint, the variable indexes or values involved in the constraint, and some other features (e.g. lower/upper bound in `domain` constraints). Also, a translation method from each cC_i to $\texttt{store}^{\mathcal{FD}}$ is defined. Third, `Space` does not support the $\texttt{store}^{\mathcal{FD}}$ clearing. Thus, to manage backtracking, `Solver` must be replaced by a new empty instance, whose cV is created as a new `IntVarArgs` containing as many new variables as the restored size of pV. Fourth, when managing a labeling, the number of variables is known and fixed. As cloning is not supported for `IntVarArgs`, a `Solver'` class is defined, similar to `Solver` but with an `IntVarArray` attribute. To manage multiple search strategies, a vector of `SearchEngine(Solver')` is created, representing `ss`. When managing a new `l_i`, the `IntVarArray` of `Solver_i'` is initialized, containing as many new variables as the current size of `Solver` cV. If a new `Solver'` is returned as solution, its `IntVarArray` is traversed, posting the equality constraints to cC and `Solver`.

3.3.2 Integration of ILOG Solver: $\mathcal{TOY}(\mathcal{FD}i)$

The main object of each IBM ILOG CP 1.6 application is a pool (`IloEnv` object), containing the rest of the application objects. Each of the three package libraries provides their own solver object, containing a store and an engine. The solver handles its own native representation for variables, constraints and search strategies, attached to the concrete solving techniques the engine apply on them. Solver, variables, constraints and search strategies are represented in ILOG Solver 6.8 via `IloSolver`, `IlcIntVar`, `IlcConstraint` and `IlcGoal` objects (respectively). ILOG Concert 12.2 provides the pool and isolates the modeling of an application from the representation/solving techniques of a concrete solver. It provides generic `IloIntVar`, `IloConstraint` and `IloGoal` objects to model the problem, as well as the vectors `IloModel` (`IloIntVar Array`) to gather the constraint network (and its associated variables). An `IloGoal` represents a single labeling, and is defined over an `IloIntVar` set. Each solver provides a method to translate the content of `IloModel` to its internal store. `IloSolver` traverses `IloModel`, generating and posting a mate `IlcConstraint` for each `IloConstraint` (as well as a mate `IlcIntVar` for each `IloIntVar` of the constraint). Also, it translates each `IloGoal` to perform the search on the mate `IlcIntVar` set. `IloModel` explicitly represents the store, as after translating its content, it provides access to native objects throughout mate modeling ones.

The scheme of Section 3.2 is instantiated to develop $\mathcal{TOY}(\mathcal{FD}i)$, with ILOG Solver 6.8 as its $\mathcal{FD}$ solver. Whereas an instance of `IloSolver` represents $\texttt{store}^{\mathcal{FD}}$ and $\texttt{engine}^{\mathcal{FD}}$, an instance of `IloModel` and `IloIntVarArray` represent cC and cV (respectively). Thus, there are three different representations for the variables and con-

straints of a goal: Prolog, Concert and Solver ones. Replication of Concert and Solver objects is mandatory because the API method for posting an `IlcConstraint` to `IloSolver` requests an `IloConstraint` previously posted to `IloModel` as an argument (an `IlcConstraint` can be posted straight to `IloSolver`, but only in the context of a search or `IlcGoal`). When backtracking is detected, $\texttt{store}^{\mathcal{FD}}$ is cleared and the whole content of `IloModel` is dumped to it (with an implicit translation to the `IloSolver` input). To specify d constraints, a new class (inheriting from `IlcConstraint`) is defined. Then, Concert templates are used to specify its mate `IloConstraint` representation and the Concert to Solver object translation.

3.4 Performance

The performance of the $\mathcal{TOY}(\mathcal{FD})$ versions is measured by solving four of the pure CP($\mathcal{FD}$) benchmarks problems available at CSPLib [103]: The CSP's *Magic Sequence*, *N-Queens* and *Langford's number*, as well as the COP of *Golomb Rulers*. N-Queens and Golomb Rulers were previously described in Sections 2.1.1 and 2.1.2, respectively. Regarding Magic Sequence, it consists of finding a sequence of integers $[v_0, \ldots, v_{n-1}]$ such that for all $0 \leq i < n$, the number i occurs exactly v_i times in the sequence. For example, the two feasible solutions of Magic-4 are $[1, 2, 1, 0]$ and $[2, 0, 2, 0]$. Regarding Langford's number, it consists of a sequence of $M \times N$ integers (ranging in $1..N$), so that each appearance of the number i is i numbers on from the last. For example, the two feasible solutions of Langford's-(2,4) are $[2, 3, 4, 2, 1, 3, 1, 4]$ and $[4, 1, 3, 1, 2, 4, 3, 2]$.

This set of problems is representative enough. First, all the problems are parametric, and thus they allow to test the performance of the $\mathcal{TOY}(\mathcal{FD})$ versions as the instances of each problem scale up. Second, they include the whole set of $\mathcal{FD}$ constraints of the $\mathcal{TOY}(\mathcal{FD})$ repertoire. More specifically:

- Langford's and Golomb include the relational constraints #>, #< and #>=, as well as the arithmetic operator #-.
- Langford's includes the propositional constraint `post_implication`.
- All the problems include explicit `domain` constraints to set initial values for their variables.
- All the problems include global constraints, with Queens, Langford's and Golomb using an `all_different` constraint, and Magic using a `distribute` one (besides also using redundant `sum` and `scalar_product` constraints to increase the propagation).
- Finally, the model of each problem ends with a `labeling` search strategy. In the case of Golomb, it includes a minimization function to perform branch and bound during search. The $\mathcal{TOY}(\mathcal{FD})$ models of the four problems are available at: `http://gpd.sip.ucm.es/ncasti/models.zip`.

Taking advantage of the scalability of the problems, for each of them three different instances have been selected, whose solving times (for all the $\mathcal{TOY}(\mathcal{FD})$ versions) are of tenths of seconds, seconds and minutes, respectively. From now on, each instance is identified by the capital letter of the name of the problem (m, q, l and g for Magic, Queens, Langford's and Golomb, respectively) and the parameter values: 400 and 900 sequence's length for Magic (not a single instance solved in minutes has been found, as $\mathcal{TOY}(\mathcal{FD})$ runs out of memory before posting the whole constraint network to `storeFD`), 90, 105 and 120 for Queens, $(2, 119)$, $(2, 127)$ and $(2, 131)$ for Langford's and 9, 10 and 11 for Golomb.

Table 3.1 presents the results. Column `Instance` represents the instance being run (the concrete $\mathcal{TOY}(\mathcal{FD})$ version solving it is represented by `FDs`, `FDg` and `FDi` for $\mathcal{TOY}(\mathcal{FD}s)$, $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$, respectively). Next block of three columns represent results for incremental propagation mode. Column `Incremental` represents the CPU solving time, measured in milliseconds (for each problem, the best search strategy between first unbound and first fail is first selected, and this this same strategy is applied to the three $\mathcal{TOY}(\mathcal{FD})$ versions). Column `Perc_I` represents the percentage of the CPU solving time devoted to $\mathcal{FD}$ search exploration. To compute it, a new variant of the model (equivalent to the original one, but containing no `labeling` primitive) is also run. Thus, by comparing the CPU times of the original and the variant models is easy to obtain the CPU time spent in performing the `labeling` primitive (i.e., the search exploration). Column `Sp-Up_I` represents the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$. Next block of three columns are the same, but for batch propagation. Finally, column `I/B` represents the speed-up of using batch mode instead of the incremental one.

Benchmarks are run in a machine with an Intel Dual Core 2.4Ghz processor and 4GB RAM memory. The OS used is Windows 7 Professional SP1 (32 bits.) The SICStus Prolog version used is 3.12.8. (including the `clpfd` library used for $\mathcal{TOY}(\mathcal{FD}s)$). Microsoft Visual Studio 2008 tools are used for compiling and linking the $\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}g)$ C++ code. Each instance has been executed five times (to avoid any particular side effect associated to the OS affecting the achieved time on a single run). Then, the best and worse times have been discarded, computing as a final result (to be displayed in the tables) the mean of the remaining three.

The results allow to draw the following conclusions:

First of all, the approach of enhancing the $\mathcal{TOY}(\mathcal{FD})$ performance by focusing on its $\mathcal{FD}$ solver is encouraging. For all the instances, $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ outperform $\mathcal{TOY}(\mathcal{FD}s)$ (with the single exception of L-119, for which $\mathcal{TOY}(\mathcal{FD}i)$ with incremental propagation is slower than $\mathcal{TOY}(\mathcal{FD}s)$). However, the improvement that $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ achieve ranges in a 1.15-3.57 times faster than $\mathcal{TOY}(\mathcal{FD}s)$, so a more detailed analysis by problems and instances is required.

For Magic and Golomb instances, the improvement of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$ remains stable as the instances scale up. For Magic instances,

Instance	Incremental	Perc_I	Sp-Up_I	Batch	Perc_B	Sp-Up_B	I/B
M-400 FDs	0.609	41.1	1.00	0.608	43.6	1.00	1.00
M-400 FDg	0.422	22.3	1.45	0.405	22.0	1.49	1.04
M-400 FDi	0.530	26.8	1.15	0.468	30.3	1.30	1.14
M-900 FDs	3.00	39.1	1.00	2.93	40.4	1.00	1.02
M-900 FDg	2.00	25.6	1.49	1.98	26.6	1.47	1.01
M-900 FDi	2.53	22.9	1.19	2.20	31.2	1.33	1.15
Q-90 FDs	0.142	45.1	1.00	0.141	67.4	1.00	1.01
Q-90 FDg	0.078	42.0	1.81	0.056	54.6	2.50	1.39
Q-90 FDi	0.110	29.1	1.30	0.078	39.7	1.81	1.41
Q-105 FDs	3.45	97.3	1.00	3.43	97.7	1.00	1.01
Q-105 FDg	1.05	96.6	3.33	1.02	97.2	3.33	1.03
Q-105 FDi	1.25	92.6	2.78	1.17	94.7	2.94	1.06
Q-120 FDs	454.43	99.9	1.00	453.18	99.9	1.00	1.00
Q-120 FDg	129.88	99.9	3.45	128.83	99.9	3.57	1.01
Q-120 FDi	154.00	99.9	2.94	153.83	99.9	2.94	1.00
L-119 FDs	0.406	42.1	1.00	0.406	50.0	1.00	1.00
L-119 FDg	0.296	47.6	1.37	0.296	58.1	1.37	1.00
L-119 FDi	0.530	26.6	0.76	0.296	52.7	1.37	1.79
L-127 FDs	12.07	97.7	1.00	12.03	98.0	1.00	1.00
L-127 FDg	4.62	96.3	2.63	4.60	96.3	2.63	1.00
L-127 FDi	4.35	88.9	2.78	4.09	95.8	2.94	1.06
L-131 FDs	251.43	99.9	1.00	250.15	99.9	1.00	1.01
L-131 FDg	98.53	99.8	2.56	97.80	99.8	2.56	1.01
L-131 FDi	87.00	99.4	2.86	86.07	99.8	2.94	1.01
G-9 FDs	0.842	98.1	1.00	0.842	98.2	1.00	1.00
G-9 FDg	0.250	98.8	3.33	0.250	93.6	3.33	1.00
G-9 FDi	0.421	97.9	2.00	0.407	96.3	2.08	1.03
G-10 FDs	7.35	99.6	1.00	7.27	99.8	1.00	1.01
G-10 FDg	2.11	99.1	3.45	2.11	99.2	3.45	1.00
G-10 FDi	3.56	99.2	2.08	3.49	99.5	2.08	1.02
G-11 FDs	153.05	99.9	1.00	151.48	99.9	1.00	1.01
G-11 FDg	42.01	99.9	3.57	41.92	99.9	3.57	1.00
G-11 FDi	72.65	99.9	2.13	72.55	99.9	2.08	1.00

Table 3.1: Performance of the Different $\mathcal{TOY}(\mathcal{FD})$ Versions

$\mathcal{TOY}(\mathcal{FD}g)$ is a 1.45-1.49 times faster than $\mathcal{TOY}(\mathcal{FD}s)$, and $\mathcal{TOY}(\mathcal{FD}i)$ 1.15-1.33 times. For Golomb instances, the improvement is even greater, with $\mathcal{TOY}(\mathcal{FD}g)$ being 3.33-3.58 times faster and $\mathcal{TOY}(\mathcal{FD}i)$ 2.00-2.13.

For Queens and Langford's instances, the improvement achieved is different for the instances solved in tenths of seconds (Q-90 and L-119) than the one for both the instances solved in seconds and minutes (Q-105, Q-120, L-127 and L-131). In Queens, whereas the improvement of $\mathcal{TOY}(\mathcal{FD}g)$ goes from 1.81-2.50 (for Q-90) to 3.33-3.57 (for Q-105 and Q-120), the improvement of $\mathcal{TOY}(\mathcal{FD}i)$ goes from 1.30-1.81 to 2.78-2.94. In Langford, whereas the improvement of $\mathcal{TOY}(\mathcal{FD}g)$ goes from 1.37 (for L-119) to 2.56-2.63 (for L-127 and L-131), the improvement of $\mathcal{TOY}(\mathcal{FD}i)$ goes from 1.37 to 2.78-2.94 (being even faster than $\mathcal{TOY}(\mathcal{FD}g)$).

Second, interestingly, when solving an instance there is a clear correlation between the time the $\mathcal{TOY}(\mathcal{FD})$ versions devotes to search exploration and the improvement $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ achieve w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$.

For Magic and Golomb instances, the improvement achieved remains stable as the instances scale up, and the time the systems devote to search exploration remain stable as well. In Magic instances, $\mathcal{TOY}(\mathcal{FD}s)$ devotes around a 40%-43%, $\mathcal{TOY}(\mathcal{FD}g)$ around a 20%-25%, and $\mathcal{TOY}(\mathcal{FD}i)$ around a 23%-30%. In Golomb instances, all the versions devote around a 98%-100%.

For Queens and Langford's instances, the percentage of CPU solving time devoted to search exploration clearly grows from Q-90 and L-119 to Q-105, Q-120, L-127 and L-131 (and the improvement achieved grows as well). Whereas for Q-90 and L-119 the percentage is small and very different for the three systems (ranging in 29%-67% for Q-90 and in 26%-58% for L-119), for Q-105, Q-120, L-127 and L-131 the percentage of each $\mathcal{TOY}(\mathcal{FD})$ version is above 90% for the other instances.

Finally, the fact that batch propagation mode devotes more time to search exploration than incremental one makes perfect sense, as batch mode starts the search by propagating for the first time the whole set of constraints posted to the solver (which in incremental mode is already done). However, the time spent in this initial propagation is smaller than propagating the constraint network incrementally, as it can be seen that batch mode is faster than incremental one for all the instances.

But, in general, the differences achieved between both propagation modes are really small, about an order of magnitude smaller than the CPU solving time of the instance. Thus, for the instances solved in minutes, the differences in $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ are smaller than a second, and about 1.0-1.5 seconds for $\mathcal{TOY}(\mathcal{FD}s)$. For the instances solved in seconds, $\mathcal{TOY}(\mathcal{FD}s)$ and $\mathcal{TOY}(\mathcal{FD}g)$ have differences ranging in 0.2-0.8 tenths of seconds, and $\mathcal{TOY}(\mathcal{FD}i)$ increase these differences to 2.3 and 2.6 tenths of seconds for M-900 and L-127, respectively. For the instances solved in tenths of seconds, $\mathcal{TOY}(\mathcal{FD}s)$ and $\mathcal{TOY}(\mathcal{FD}g)$ match both propagation mode times, but $\mathcal{TOY}(\mathcal{FD}i)$ requires 3.3 tenths of seconds more for solving L-119 with incremental mode, which is nearly the double of the CPU time spent with batch mode (and, once

again, even 0.1 seconds slower than $\mathcal{TOY}(\mathcal{FD}s)$).

3.5 Related Work

First, as a CFLP($\mathcal{FD}$) system, $\mathcal{TOY}(\mathcal{FD})$ is positioned w.r.t. to other existing multi-paradigm declarative programming systems:

As mentioned in Section 2.3.5, the most related approach to our framework is Curry [90], and more specifically its PAKCS implementation [93]: Both $\mathcal{TOY}(\mathcal{FD})$ and PAKCS provide pure FLP languages, including characteristics of pure LP and FP paradigms. The operational semantics of both systems is based on lazy narrowing, integrating the LP and FP solving mechanisms of unification and rewriting, respectively. Also, both systems are implemented in Prolog (PAKCS also includes a SWI-Prolog back-end) and their programs are compiled to Prolog. Finally, both $\mathcal{TOY}(\mathcal{FD}s)$ and PAKCS rely on SICStus `clpfd` for $\mathcal{FD}$ constraint solving. A detailed comparison about modeling and solving CSP's and COP's in PAKCS and $\mathcal{TOY}(\mathcal{FD})$ is presented in Chapters 7 and 8, respectively.

Another CFLP($\mathcal{D}$, $\mathcal{S}$, $\mathcal{L}$) scheme is proposed in [130]. It formalizes a family of languages parameterized by a constraint domain $\mathcal{D}$, a strategy $\mathcal{S}$ which defines the cooperation of several constraint solvers over $\mathcal{D}$, and a constraint lazy narrowing calculus $\mathcal{L}$ for solving constraints involving functions defined by user given constrained rewriting rules. This approach relies on solid work on higher-order lazy narrowing calculi and has been implemented on top of Mathematica [131]. However, its main limitation (compared to $\mathcal{TOY}(\mathcal{FD})$) is the lack of a declarative semantics.

Also, the system *Oz* [194] provides salient features of FP such as compositional syntax and first-class functions, and features of LP and CP($\mathcal{FD}$) including logic variables and constraints. The Mozart Programming System [139] is the primarily implementation of Oz. However, it is quite different to $\mathcal{TOY}(\mathcal{FD})$, as it generalizes the CLP($\mathcal{FD}$) and concurrent CP($\mathcal{FD}$) paradigms (and thus its computation mechanism is not based on lazy narrowing). Functions and constraints are not really integrated, in the sense that they do not have the same category. It supports a class of lazy functions (based on a demand-driven computation), but this is not an inherent feature of the language (functions have to be made lazy explicitly via the concept of futures).

Second, the approach of interfacing external state-of-the-art CP($\mathcal{FD}$) solvers to improve the solving performance of $\mathcal{TOY}(\mathcal{FD})$ has been also used by other (mono-paradigm) declarative programming systems:

The LP language Mercury [184] is considerably faster than traditional Prolog implementations, but lacks support for full unification. The CLP($\mathcal{FD}$) system HAL [62] compiles to Mercury, and it is specifically designed to support the construction of and experimentation with constraint solvers. It provides a well-defined solver interface, mutable global variables for implementing a constraint store, and dynamic schedul-

ing for combining, extending and writing new constraint solvers. It also allows to call solvers written in other languages (e.g. C) with little overhead, as well as to "plug and play" with different constraint solvers over the same domain.

The literature proposed numerous CLP($\mathcal{FD}$) approaches, as SICStus Prolog, SWI-Prolog, GNU Prolog [5], Ciao Prolog [4] and B-Prolog [31]. All these systems are based on Prolog and, as $\mathcal{TOY}(\mathcal{FD}s)$, they provide a clean and elegant interface to their native Prolog $\mathcal{FD}$ constraint solver library. Thus, as already mentioned, the scheme presented in Section 3.2 can be easily adaptable to them. For example, the new version of ECLiPSe [16] interfaces Gecode as its $\mathcal{FD}$ solver.

Finally, the Monadic Constraint Programming (MCP) [169] framework illustrates how the abstractions and mechanisms from FP such as monads, higher-order functions, continuations and lazy evaluation are valuable notions for defining and building CP($\mathcal{FD}$) systems. The FD-MCP framework [205] supports the integration of CP($\mathcal{FD}$) solvers in Haskell. It introduces an extra layer between the language and the concrete CP($\mathcal{FD}$) solver being used, in order to support the integration of different solvers. The current version of the system includes an interface to Gecode, which can be accessed either directly from the system (intermixing constraint solving with rewriting), or by compiling the Haskell model to a C++ Gecode one.

3.6 Conclusions

Within CP($\mathcal{FD}$) systems, the functional-logic language provided by $\mathcal{TOY}(\mathcal{FD})$ represents an appealing approach for modeling CSP's and COP's. First, its declarative nature abstracts the problem specification, allowing the user to model a problem by simply describing the properties its solutions must hold. This represents an advantage w.r.t. imperative languages (as the ones of C++ CP($\mathcal{FD}$) systems), where the way the model is built is procedural (although the paradigm is still declarative, i.e., the C++ objects are used to describe a declarative model). Second, its general purpose nature allows the integration of the model into larger applications, in contrast to specific-purpose languages (as the ones of algebraic CP($\mathcal{FD}$) systems). Third, its high expressiveness includes the main features from both logic and functional languages (as the ones of CLP($\mathcal{FD}$) and CFP($\mathcal{FD}$) systems, respectively), and its $\mathcal{FD}$ constraint catalogue includes relational, arithmetic, propositional, domain and global constraints.

Unfortunately, the solving performance of $\mathcal{TOY}(\mathcal{FD})$ is slightly penalized by the inherent overhead coming from the integration of a CP($\mathcal{FD}$) solver within its operational semantics. However, due to the combinatorial nature of the CSP's and COP's being tackled, it is expected that, as long as the instances scale up enough, most of the CPU time for solving them is spent in $\mathcal{FD}$ solver computations. Thus, a suitable approach for improving the system performance is by replacing its $\mathcal{FD}$ solver (currently based on the underlying SICStus `clpfd`, and thus naming the system version as $\mathcal{TOY}(\mathcal{FD}s)$)

by the state-of-the-art C++ CP($\mathcal{FD}$) solvers of Gecode and ILOG Solver. That is, keeping the same $\mathcal{TOY}(\mathcal{FD})$ model for solving a problem (specifically, keeping the same $\mathcal{FD}$ constraint network and search strategy), and rely on new solvers capable of solving it faster.

This chapter has presented the architecture of $\mathcal{TOY}(\mathcal{FD})$, using an example to describe how goal computations are achieved by relying both on symbolic computation and $\mathcal{FD}$ constraint solving. It has focused on the interface between the system and its $\mathcal{FD}$ solver, which is implemented as a set of Prolog predicates. It has identified the commands the system requires to coordinate the solver, and the posting of a concrete constraint has been used to describe this coordination.

Then, a scheme for interfacing C++ CP($\mathcal{FD}$) solvers into $\mathcal{TOY}(\mathcal{FD})$ has been presented (which can be easily adapted to other CLP($\mathcal{FD}$) or CFLP($\mathcal{FD}$) systems implemented in Prolog). It has focused on the extra difficulties arisen in terms of communication with the solver and different variables, constraints and types representations. Also, it has described how to adapt a C++ CP($\mathcal{FD}$) solver to the CFLP($\mathcal{FD}$) requirements of model reasoning, multiple search strategies (interleaved with constraint posting) and both incremental and batch propagation modes. The scheme has been shown to be generic enough, interfacing Gecode and ILOG Solver (leading to the new versions $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$, respectively) by finding no extra interface difficulties but the ones described in the scheme.

Finally, the performance of the three $\mathcal{TOY}(\mathcal{FD})$ versions has been measured by using a subset of problems from the classical CP($\mathcal{FD}$) benchmarks of CSPlib, complete enough as it contains three CSP's and a COP, also covering the whole repertoire of $\mathcal{FD}$ constraints supported by $\mathcal{TOY}(\mathcal{FD})$. It has been shown that $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ outperform $\mathcal{TOY}(\mathcal{FD}s)$, but the improvement achieved (ranging in 1.15-3.57 times faster) is dependent on the concrete problem and instance solved. There is a clear correlation between the percentage of CPU solving time devoted to search exploration and the improvement achieved. This turns the performance of $\mathcal{TOY}(\mathcal{FD})$ into purely CP($\mathcal{FD}$) dependent. That is, the CPU time of each version directly comes from the performance of its concrete solver in achieving the pure CP($\mathcal{FD}$) mechanism of performing a search exploration by propagating basic and global constraints. Thus, the next step to improve the $\mathcal{TOY}(\mathcal{FD})$ performance is to focus on the search strategy, enhancing the capabilities of the language to specify *ad hoc* strategies requiring less search exploration to find solutions.

Chapter 4

Developing new Search Strategies

The use of *ad hoc* search strategies has been identified as a key point for solving CSP's and COP's, as they exploit the knowledge about the structure of the problem and its solutions. The performance experiments on classical CP($\mathcal{FD}$) benchmarks (cf. Section 3.4) have shown that, as long as the instances of these problems scale up enough, most of their CPU solving time is devoted to search exploration. Thus, following again the Amdahl's law, a suitable approach to improve the solving efficiency of $\mathcal{TOY}(\mathcal{FD})$ is by modifying the models for the problems, replacing the basic `labeling` search strategies by *ad hoc* ones.

CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems as $\mathcal{TOY}(\mathcal{FD})$ provide a declarative view of a search strategy specification, in contrast to the procedural view of C++ CP($\mathcal{FD}$) libraries as Gecode or ILOG Solver (which make the programming of a strategy to depend on low-level details associated to the constraint solver, and even on the concrete machine the search is being performed). Also, due to their model reasoning capabilities, $\mathcal{TOY}(\mathcal{FD})$ treats search primitives as simple expressions, making possible to place a search primitive at any point of the program, combine several primitives to develop complex search heuristics, intermix search primitives with constraint posting, and use indeterminism to apply different search scenarios for solving a problem.

The main contribution of this chapter is to present a set of new parametric search primitives for the $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ versions previously developed (in a setting easily adaptable to other CLP($\mathcal{FD}$) or CFLP($\mathcal{FD}$) systems implemented in Prolog and interfacing external C++ CP($\mathcal{FD}$) solvers). The motivation of this approach is to take advantage of both the high expressivity of $\mathcal{TOY}(\mathcal{FD})$ for specifying search strategies, and of the high efficiency of Gecode and ILOG Solver. Thus, the $\mathcal{TOY}(\mathcal{FD})$ language is enhanced for $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ with new parametric search primitives, implementing them in Gecode and ILOG Solver by extending their underlying

search libraries. Besides that, the two other search approaches already supported by $\mathcal{TOY}(\mathcal{FD}s)$, $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ remain available. The first one consists of defining a new search from scratch at $\mathcal{TOY}(\mathcal{FD})$ level, relying on reflection functions (which retrieve information about the constraint store at the particular moment of their execution) to represent the search procedure. The main drawback of reflection functions is that they have associated a notion of state, thus making their result dependent of their execution order (and breaking the declarative nature of $\mathcal{TOY}(\mathcal{FD})$ programs). The second search approach consists of using the search primitive `labeling` (simply relying on the predefined search strategies already existing in SICStus `clpfd`, Gecode and ILOG Solver, respectively).

The chapter is organized as follows: Section 4.1 presents an abstract description of the new parameterizable $\mathcal{TOY}(\mathcal{FD})$ search primitives, pointing out some novel concepts not directly available neither in Gecode nor in ILOG Solver. Also, it points out how to specify some search criterion at $\mathcal{TOY}(\mathcal{FD})$ level and how easily the strategies can be combined to set different search scenarios. Section 4.2 describes the implementation of the primitives, presenting first an abstract view of the $\mathcal{TOY}(\mathcal{FD})$ requirements, and how they are deployed on the Gecode and ILOG Solver libraries. It also evaluates the impact of the search strategies implementation in the architecture of the system. Section 4.3 analyzes the new $\mathcal{TOY}(\mathcal{FD})$ performance achieved. The benchmark set of CP($\mathcal{FD}$) problems of Section 3.4 is revisited, showing that the use of the search strategies improve the solving performance of both $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$. Section 4.4 presents some related work. Finally, Section 4.5 reports conclusions.

4.1 Search Primitives Description

This section presents eight new $\mathcal{TOY}(\mathcal{FD})$ primitives for specifying search strategies, allowing the user to interact with the solver in the search for solutions. These primitives bridge the gap between the other two classical approaches available in $\mathcal{TOY}(\mathcal{FD})$: Defining a whole search procedure at $\mathcal{TOY}$ level (by using reflection functions), and relying on the set of predefined search strategies available in the solver library. Each primitive has its own semantics, and it is parameterizable by several basic components. As described in Section 3.2.4, the search primitives are considered by the language as simple expressions, so intermixing search strategies with the regular posting of constraints is allowed. The section describes the primitives and their components (including its type declaration) from an abstract (solver independent) point of view. It emphasizes some novel search concepts arisen, which are not available in the predefined search strategies of Gecode and ILOG Solver. It also shows how easy and expressive it is to specify some search criterion at $\mathcal{TOY}(\mathcal{FD})$ level, and the appealing possibilities $\mathcal{TOY}(\mathcal{FD})$ offers to apply different search strategies for solving a CP($\mathcal{FD}$) problem.

4.1.1 Labeling Primitives

In this section four search primitives are described: `lab`, which represents a variation of the classical `labeling`, including new variable and value selection criterion, as well as the possibility of labeling just a subset of the variables involved. `labB`, which represents a variation of `lab`, but exploring just a branch of the search tree (i.e., leading to a first failed or solution node, and then stopping the search, with no backtracking allowed). `labW`, which performs an exhaustive breadth exploration of some levels of the search tree, further sorting the satisfiable solutions by a specified criterion. `labO`, which represents the optimization variation of `lab`.

Primitive lab

`lab:: varOrd -> valOrd -> int -> [int] -> bool`

This primitive collects (one by one) all possible combinations of values satisfying the set of constraints posted to the solver. It is parameterized by four basic components. The first and second ones represent the variable and value order criterion to be used in the search strategy, respectively. To express them the enumerated datatypes `varOrd` and `valOrd` have been defined in $\mathcal{TOY}$, covering all the predefined criterion available in the Gecode documentation [173]. They also include a last case (`userVar` and `userVal`, respectively) in which the user implements its own variable/value selection criterion at $\mathcal{TOY}(\mathcal{FD})$ level. The third element `N` represents how many variables of the variable set are to be labeled. This represents a novel concept which is not available in the predefined search strategies of Gecode and ILOG Solver. The fourth argument represents the variable set `S`. Thus, the search heuristic labels just the first `N` variables of `S` being selected by the `varOrd` criterion.

Figure 4.1 presents a $\mathcal{TOY}(\mathcal{FD})$ program (top) and goal (bottom) showing how expressive, easy and flexible it is to specify a search criterion in $\mathcal{TOY}(\mathcal{FD})$. In the example, the search strategy of the goal uses the `userVar` and `userVal` selection criterion, specified by the user in the functions `myVarOrder` and `myValOrder`, respectively (as it can be seen, in this setting just one user variable and value selection criterion per program is allowed).

The `lab` search strategy is applicable to the constraint network posted by the $\mathcal{TOY}(\mathcal{FD})$ goal `domain [X,Y,Z] 0 4, Y /= 1, Y /= 3, Z /= 2`. Then, the computation continues by processing `rest of goal` for each feasible solution found by the `lab` strategy. It acts over the set of variables `[X,Y,Z]`, but it is only expected to label two of them.

The function `myVarOrder` selects first the variable with more intervals in its domain. It receives the list of variables involved in the search strategy, returning the index of the selected one. To do so it uses the auxiliary functions `from` and `cmp`, the predefined functions `fst`, `foldl`, `zip`, `take`, `length`, `map`, `head`, `last` and `(.)` (all of them with an equivalent semantics as in Haskell), and, finally, the reflection func-

```
include "cflpfd.toy"
%
myVarOrder:: [int] -> int
myVarOrder V = fst (foldl cmp (0,0)
                              (zip (take (length V) (from 0))
                                   (map (length . get_dom) V)))
%
myValOrder:: [[int]] -> int        | from:: int -> [int]
myValOrder D = head (last D)       | from N = [N | from (N+1)]
%
cmp:: (int,int) -> (int,int) -> (int,int)
cmp (I1,V1) (I2,V2) = if (V1 >= V2) then (I1,V1) else (I2,V2)
-----------------------------------------------------------------
TOY(FD)> domain [X,Y,Z] 0 4, Y /= 1, Y /= 3, Z /= 2,
         lab userVar userVal 2 [X,Y,Z], ... (rest of goal)
```

Figure 4.1: Variable and Value User-Defined Criterion

tion `get_dom`, which accesses the internal state of the solver to obtain the domain of a variable (this domain is presented as a list of lists, where each sublist represents an interval of values).

The function `myValOrder` receives as its unique argument the domain of the variable, returning the lower bound of its upper interval. So, in conclusion, the first two (partial) solutions the `lab` strategy leads to are: `{X in 0..4, Y -> 4, Z -> 3}` and `{X in 0..4, Y -> 4, Z -> 4}`.

Primitive labB

`labB:: varOrd -> valOrd -> int -> [int] -> bool`

This primitive uses the same four basic elements as `lab`. However, its semantics is different, as it follows the `varOrd` and `valOrd` criterion to explore just one branch of the search tree, with no backtracking allowed. The Queens problem is used to explain this behavior.

Using `lab unassignedLeftVar averageVal 2 [X1,X2,X3,X4]` two solutions are found: `{X1 -> 2, X2 -> 4, X3 -> 1, X4 -> 3}` and `{X1 -> 3, X2 -> 1, X3 -> 4, X4 -> 2}` (cf. Figure 2.1 for a graphical representation of both solutions). If `labB unassignedLeftVar averageVal 2 [X1,X2,X3,X4]` is used, then only the first solution `{X1 -> 2, X2 -> 4, X3 -> 1, X4 -> 3}` is found, as, after exploring the successful search path, no backtracking is allowed.

Moreover, if `labB unassignedLeftVar smallestVal 2 [X1,X2,X3,X4]` is used, then the strategy fails, getting no solutions. Figure 4.2 (4×4 square `Board` and tree) shows the computation process. First, the selected criterion assigns `X1 -> 1` at root node (1), leading to node 2. Propagation reduces search space to `(X2 in 3..4, X3 in 2` $\vee$ `4, X4 in 2..3)`, pruning nodes 3 and 4. Then, computation assigns

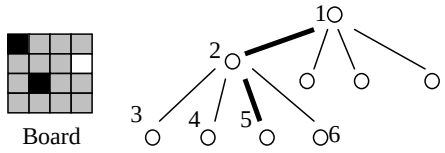

Figure 4.2: Applying labB to the Queens problem

X2 -> 3 (leading to node 5), and propagation leads to an empty domain for X3. So, the explored tree path leads to no solutions. As it can be seen, propagation during search modifies the intended branch to be explored (in the goal example, it explores the branch 1-2-5 instead of the 1-2-3).

Primitive labW

`labW:: varOrd -> bound -> int -> [int] -> bool`

This primitive performs an exhaustive breadth exploration of the search tree, storing the satisfiable leaf nodes achieved to further sort them by a specified criterion. The first parameter represents the variable selection criterion (no value selection is necessary, as the search would be exhaustive for all the values of the selected variables). The second parameter represents the *best* node selection criterion. To express it in $\mathcal{TOY}(\mathcal{FD})$, the enumerated datatype `ord` has been defined, taking as possible values the smallest/largest remaining search space of the product cardinalities of the labeling/solver-scope variables. Again, a last case (`userBound`) allows to specify the bound criterion at $\mathcal{TOY}(\mathcal{FD})$ level. The third parameter sets the breadth level of exhaustive exploration of the tree. Finally, as usual, the last parameter is the set of variables to be labeled.

A first example is considered to understand the semantics of `labW`. Figure 4.3 presents a $\mathcal{TOY}(\mathcal{FD})$ goal with four variables, where two implication constraints relate X and Y with V1 and V2, respectively.

If a `lab unassignedLeftVar smallestVal 2 [X,Y,V1,V2]` strategy had been used to label the first two unbound vars of `[X,Y,V1,V2]`, then the search would have explored the search tree obtaining (one by one) the next four satisfiable nodes: `{X -> 0, Y -> 0}`, `{X -> 0, Y -> 1}`, `{X -> 1, Y -> 0}` and `{X -> 1, Y -> 1}`. As it can be seen, whereas the first solution computed by `lab` leads to compute the "rest of goal" from a 12 candidates search space, the third solution leads to a 6 candidates one.

Figure 4.4 represents the exploration if a `labW unassignedLeftVar smallestSearchTree 2 [X,Y,V1,V2]` strategy is used. An horizontal black line represents the depth the tree is explored, each black node represents a solution, and the triangle each node has below represents the remaining size of the search space (product of cardinalities of V1 and V2). The primitive `labW` explores exhaustively the search tree in breadth, storing in a data structure DS each satisfiable node. Once the tree has been completely

```
TOY(FD)> domain [X,Y] 0 1, post_implication X (#=) 1 V1 (#>) 1,
         domain [V1,V2] 0 3, post_implication Y (#=) 0 V2 (#>) 0,
         labW unassignedLeftVar smallestSearchTree 2 [X,Y,V1,V2],
         ... (rest of goal)
```

Figure 4.3: `labW` Example

explored, the satisfiable nodes are obtained (one by one) by using a criterion to select and remove the *best* node from DS. In the example, the selected criterion `smallestSearchTree` selects first the nodes with smallest product of cardinalities of V1 and V2 (returning first the solution of the 6 candidates). The order in which the `labW` strategy of the goal delivers the satisfiable nodes is presented in Figure 4.4.

Figure 4.5 presents a $\mathcal{TOY}(\mathcal{FD})$ program (top) and goal (bottom) with a bound criterion specified in the user function `myBound`. The *best* node procedure selection traverses all the nodes in DS, selecting first the one with minimal stored value. Thus, the user criterion specified in `myBound` assigns to each node (minus) the number of its singleton value search variables. Once again, the function `myBound` also relies on auxiliary, prelude and reflection functions. The first two solutions are `(X -> 1, Y -> 1, A -> 0, B -> 0, C -> 0)` and `(X -> 2, Y -> 1, A in 0..1, B -> 0, C -> 0)`, respectively.

In summary, `labW` represents a novel concept which is not available in the predefined search strategies of Gecode and ILOG Solver. However, it must be used carefully, as exploring the tree very deeply can lead to a explosion of satisfiable nodes, producing memory problems for DS and becoming very inefficient (due to the time spent on exploring the tree and selecting the *best* node).

Primitive labO

`labO::optType -> varOrd -> valOrd -> int -> [int] -> bool`

This primitive performs a standard optimization labeling. The first parameter

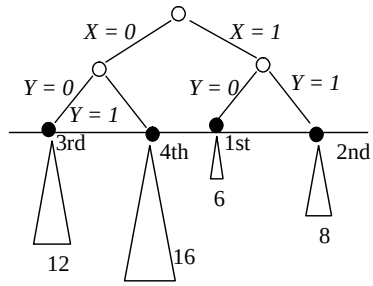

Figure 4.4: `labW` Search Tree Exploration

```
include "cflpfd.toy"
%
isBound:: [[int]] -> bool
isBound [[A,A]] = true
isBound [[A,B]] = false <== B /= A
isBound [[A,B] | RL] = false <== length RL > 0
%
myBound:: [int] -> int
myBound V = - (length (filter isBound (map get_dom V)))
-----------------------------------------------------------------
TOY(FD)> domain [X,Y] 1 2, domain [A,B,C] 0 5,
         A #< X, B #< Y, C #< Y,
         labW unassignedLeftVar userBound 2 [X,Y,A,B,C]
```

Figure 4.5: Bound User-Defined Criterion

optType contains the optimization type (minimization/maximization) and the variable to be optimized. The other four parameters are the same as in the lab primitive.

4.1.2 Fragmentize Primitives

```
frag:: domFrag -> varOrd -> intervalOrd -> int -> [int] -> bool
fragB:: domFrag -> varOrd -> intervalOrd -> int -> [int] -> bool
fragW:: domFrag -> varOrd -> bound -> int -> [int] -> bool
fragO:: domFrag->optType->varOrd->intervalOrd->int->[int]->bool
```

These four new primitives are mate to the lab* ones (where the * stands for any of them), but each variable is not labeled (bound) to a value, but *fragmented* (pruned) to a subset of the values of their domain. An introductory example is used to motivate the usefulness of these new primitives. On it a goal contains V variables and C constraints, with V' $\equiv$ {V1, V2, V3} a subset of V. The constraint domain V' 1 9 belongs to C. No constraint of C relates the variables of V' by themselves, but some constraints relate V' with the rest of variables of V.

Figure 4.6 presents the search tree exploration achieved by frag* and lab* search primitives, respectively. In the case of frag*, the domain of each variable of V' is fragmented into three intervals: A first one with the values 1, 2 and 3. A second one with the values 4, 5 and 6. A third one with the values 7, 8 and 9. Due to fragmenting the domain of each variable (instead of labeling it), frag* leads to exponentially less nodes (27) than lab* (729). On the one hand, if it is known that there is only one solution to the problem, the probabilities of finding the right combination of V' values is thus greater in frag* than in lab*. On the other hand, by assigning variables to single values (instead of pruning their domains to ranges of values), the remaining search space achieved by a lab* strategy is expected to be smaller than the one achieved by a

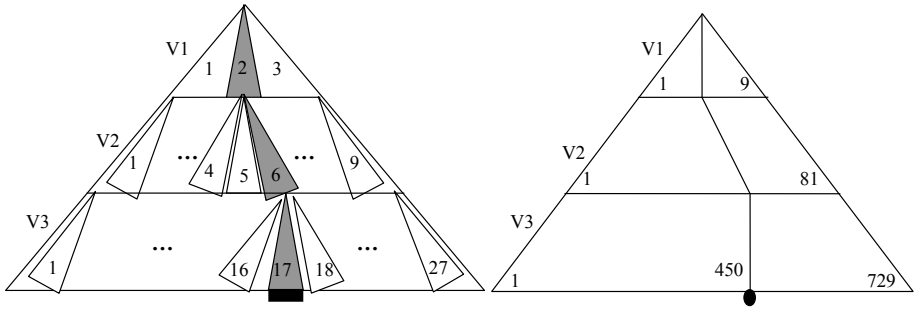

Figure 4.6: `frag` vs `lab` Search Tree

`frag*` strategy. Thus, the `frag*` search strategies can be seen as a more conservative technique, where there are less expectations of highly reducing the search space, as variables are not bound, but there is more probability of choosing a subset containing values leading to solutions (in what can be seen as a sort of generalization of *first-fail*). Coming back to the definition of each `frag*` primitive, two main differences arise w.r.t. its mate `lab*` primitive: First, it contains as an extra basic component (first argument) the datatype `domFrag`, which specifies the way the selected variable is fragmented. The user can choose between `partition n` and `intervals`. The former fragments the domain values of the variable into n subsets of the same cardinality. The latter looks for already existing intervals on the domain of the variables, splitting the domain on them. For example, in the goal `domain [X] 0 16, X /= 9, X /= 12` whereas applying `partition 3` to X fragments the domain in the subsets S1 $\equiv$ {0...4}, S2 $\equiv$ {5...8}$\cup${10} and S3 $\equiv$ {11}$\cup${13...16}, applying `intervals` fragments the domain in the subsets S1' $\equiv$ {0...8}, S2' $\equiv$ {10...11} and S3' $\equiv$ {13...16}. As a second difference, it contains an enumerated datatype `intervalOrd` (replacing the `lab*` argument `valOrd`), to specify the order in which the different intervals should be tried: First left, right, middle or random interval.

In summary, it is claimed that `frag*` primitives are a remarkable tool, to be taken into account in the context of search strategies as an alternative or a complement to the use of exhaustive labelings. Also, its use in $\mathcal{TOY}(\mathcal{FD})$ represents a novel concept which is not available in the predefined search strategies of Gecode and ILOG Solver.

4.1.3 Applying Different Search Scenarios

$\mathcal{TOY}(\mathcal{FD})$ supports non-deterministic functions, with possibly several reductions for given, even ground, arguments. The rules are applied following their textual order, and both failure and user request for a new solution trigger backtracking to the next unexplored rule. In this setting, different search strategies can be sequentially applied for solving a CP($\mathcal{FD}$) problem. For example, after posting V and C to the solver, the $\mathcal{TOY}(\mathcal{FD})$ program (top) and goal (bottom) presented in Figure 4.7 uses the non-

```
include "cflpfd.toy"
%
f:: [int] -> bool
f [V1,V2,V3] = true <==
  fragB (partition 4) unassignedLeftVar random 0 [V1],
  labB unassignedLeftVar smallestVal 0 [V2,V3]
f [V1,V2,V3] = true <==
  fragW (partition 4) unassignedLeftVar smallestTree 0 [V1],
  labB unassignedLeftVar smallestTotalVars 0 [V2,V3]
f [V1,V2,V3] = true
-----------------------------------------
TOY(FD)> ... (rest of goal, posting the constraint network
              C on the variable set V = [V1,V2,V3,...,Vk]),
         f [V1,V2,V3], lab userVar userVal 0 V
```

Figure 4.7: Applying Different Search Strategies

deterministic function `f` to specify three different scenarios for solving the goal described in Section 4.1.2. Each scenario ends with an exhaustive labeling of the set of variables V. However, the search space s this exhaustive labeling has to explore can be highly reduced by the previous evaluation of `f`.

Scenario 1: The first rule of `f` performs the search heuristic h_1 over V' $\equiv$ {V1,V2,V3}. h_1 fragments the domain of V1 into 4 subsets, selecting one randomly. If propagation succeeds, then h_1 bounds V2 and V3 to their smallest value. If propagation succeeds (with a remaining search space s_1), then h_1 succeeds, and the exhaustive labeling explores s_1. If propagation fails in one of those points, or the exhaustive labeling does not find any solution in s_1, then h_1 completely fails (as well as the first rule of `f`), as both the `labB` and `fragB` primitives just explore one branch.

Scenario 2: The second rule of `f` is tried, performing the heuristic h_2 over V'. Here a `fragW` primitive is first applied. So, if further either `labB` of h_2 or the exhaustive `lab` (acting over s_2) fails, backtracking is performed over `fragW`, providing the next *best* interval of V1 (according to the smallest search tree criterion, as in Figure 4.4). If, after trying all the intervals a solution is not found, then h_2 completely fails (as well as the second rule of `f`).

Scenario 3: If both h_1 and h_2 fail, the third rule of `f` trivially succeeds, and the exhaustive labeling is performed over the original search space obtained after posting V and C to the solver.

4.2 Search Primitives Implementation

To integrate the eight new search primitives into $\mathcal{TOY}(\mathcal{FD})$, the scheme of Section 3.2.4 is reused. Obviously, it is slightly modified for supporting the `store`$^{\mathcal{FD}}$ synchro-

nization with the solution found by engine_ss$_i$ for a frag* strategy. In this case cV is traversed, requesting engine_ss$_i$ for the lower and upper bounds of each cV$_i$, and posting the associated #>= and #<= constraints to cC.

Besides that, the new search primitives rely on the Gecode and ILOG Solver underlying search mechanisms. First, an abstract specification of the requirements the new $\mathcal{TOY}(\mathcal{FD})$ search strategies must fulfill is presented. Then, it is described how to adapt those requirements to Gecode and ILOG Solver. The modified versions of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ supporting these search strategies are available at: http://gpd.sip.ucm.es/ncasti/TOY(FD).zip

4.2.1 Abstract Specification of the Search Strategy

A single entry point (C++ function) for the different primitives is specified. Its proposed algorithm is parameterizable by the primitive type and its basic components. It is described as follows:

1. The algorithm explores the tree by iteratively selecting a variable var and a value v, creating two options: (a) Post var == v. (b) Post var /= v to continue the exploration taking advantage of the previously explored branch, recursively selecting another value to perform again (a) and (b).
2. For frag* strategies it selects an interval i instead of a value, posting in (a) both var #>= i.min and var #<= i.max. However, the (b) branch cannot take advantage by posting var #< i.min and var #> i.max, as the constraint store would become inconsistent. Thus, (b) just removes i from the set of intervals, and continue the search by selecting a new interval.
3. For labB and fragB strategies, only the (a) option is tried.
4. For lab0 and frag0 strategies, branch and bound techniques are used to optimize the search.
5. Specific functions are devoted to variable and value/interval selection strategies, as well as to the bound associated to a particular solution found by labW and fragW. Those functions include the possibility of accessing Prolog, to follow the criterion the user has specified at $\mathcal{TOY}(\mathcal{FD})$ level (using $\mathcal{TOY}(\mathcal{FD})$ functions compiled to mate Prolog predicates).
6. The primitives labW and fragW perform the breadth exploration of the upper levels of the search tree, storing all the satisfiable leaf nodes to further give them (one by one) on demand. Thus, ss contains an entity performing the search and a vector DS (cf. Section 4.1.2) containing the solutions. Each solution must be synchronized from ss to the main constraint solver. Also, a status indicates whether the exploration has finished or not.

7. The algorithm finishes (successfully) when it finds a solution, except for `labW` and `fragW` strategies, where it stores the solution node and triggers an explicit failure, continuing the breadth exploration of the tree.

8. A counter is used to control that only the specified amount of variables of the variable set is labeled/pruned.

The next two sections adapt this specification to Gecode and ILOG Solver, respectively. Table 4.1 summarizes the different notions provided by both libraries.

4.2.2 Gecode

Search strategies in Gecode are specified via `Branchers`, which are applied to the constraint solver (`Space`) to define the shape of the search tree to be explored. The `Space` is then passed to a `Search Engine`, whose execution method looks for a solution by performing a depth-first search exploration of the tree. This exploration is based on cloning `Spaces` (two `Spaces` are said to be equivalent if they contain equivalent stores) and hybrid recomputation techniques to optimize the backtracking. As `Spaces` constitute the nodes of the search tree, a solution found by the `Search Engine` is a new `Space`. The library allows to create a new class of `Brancher` by defining three class methods: `status`, which specifies if the current node is a solution, or their children must be generated to continue with their exploration. `choice`, which generates an object `'o'` containing the number of children the node has, as well as all the necessary information to perform their exploration. `commit`, which receives `o` and the concrete children identifier to perform its exploration (generating a new `Space` to be placed at that node).

Adaptation to the Specification. The search strategies are implemented via two layers. First, a new class of `Brancher MyGenerate`, which carries out the tree exploration by the combination of the `status`, `choice` and `commit` methods. As each node of the tree is a `Space`, the methods are applied to it. Second, a `Search Engine`, controlling the search by receiving the initial `Space` and making the necessary clones to traverse the tree. In this setting, the abstract description presented before is instantiated to Gecode as follows:

1. The `choice` method deals with the selection of the variable `var` and the value `v`, creating an object `o` with them as parameters, as well as the notion of having two children. The variable selection must rely on an external register `r`, being controlled by the `Search Engine` and thus independent on the concrete node (`Space`) the `choice` method is working with. The register is necessary to ensure that, whether a father generates its right hand child by posting `var /= v`, this child will reuse `r` to select again `var` (as a difference to the left hand child, which removes the `r` content to select a new variable).

Search Concept	Gecode	ILOG Solver
Search trigger	`Search Engine`	`IloGoal` stack
Tree node	`Space`	`IloGoal` attributes
Node exploration	`Brancher Commit`	`IloGoal` execution
Child generation	`Brancher Choice`	`IloGoal` constructor
Solution check	`Brancher Status`	Stack with no `IloAnd`
Solution abstraction	`Space`	Tree path (var,value) vector

Table 4.1: Different Search Concept Abstractions in Gecode and ILOG Solver

2. For `frag*` strategies, instead of passing `val` to o, the `choice` method generates a vector with all the different intervals to be tried, and the size of this vector is passed as its number of children.
3. For `labB` and `fragB`, only one child is considered.
4. For `labO` and `fragO`, a specialized branch and bound `Search Engine` provided by Gecode is used.

6 The search entity is the `Search Engine` and the notion of solution is a `Space`.

7 For `labW` and `fragW`, the `Search Engine` uses a loop, requesting solutions one by one until no more are found (the breadth exploration of the search tree has finished).

8 Only the left hand child of `lab*` strategies increments the counter value, and the `status` method checks the counter to stop the search at the precise moment.

4.2.3 ILOG Solver

Search strategies in ILOG Solver are performed via the execution of `IloGoals`. An `IloGoal` is a daemon characterized by its constructor and its execution method. The constructor creates the goal, initializing its attributes. The execution method triggers the algorithm to be processed by the constraint solver (`IloSolver`), and can include more calls to goal constructors, making the algorithm processed by `IloSolver` to be the consequence of executing several `IloGoals`. An `IloGoal` fails if `IloSolver` becomes inconsistent by running its execution method; otherwise the goal succeeds. The library allows to create a new class of `IloGoal` by defining its constructor and execution method. Four basic goal classes are provided for developing new goals with complex functionality. Goals `IlcGoalTrue` and `IlcGoalFalse` make the current goal succeed and fail, respectively. Goals `IlcAnd` and `IlcOr`, both taking two subgoals as arguments, make the current goal succeed depending on the behavior of its subgoals. While `IlcAnd` succeeds only if its two subgoals succeed, `IlcOr` creates a restorable

choice point which executes its first subgoal, restores the solver state at the choice point on demand, and executes its second subgoal.

Adaptation to the Specification. The search strategies are implemented via the new `IloGoal` classes `MyGenerate` and `MyInstantiate`. Whereas the former deals with the selection of a variable, the latter deals with its binding/prunning to a value/interval. In this setting, the abstract description presented before is instantiated to ILOG Solver as follows:

1. The control of the tree exploration is carried out by `MyGenerate`, which selects a variable and uses the recursive call `IlcAnd(MyInstantiate, MyGenerate)` to bind it and further continuing processing a new variable. In `MyInstantiate`, the alternatives (a) and (b) are implemented with an `IlcOr(var == val, IlcAnd( var /= var, MyInstantiate))`.
2. It dynamically generates a vector with the available intervals on each different `MyGenerate` call.
3. Only the goal `var == val` is tried.
4. The branch and bound is explicitly implemented. Thus, before selecting each new variable, it is checked if the current optimization variable can improve the bound previously obtained; otherwise an `IloGoalFail` is used to trigger backtracking (as well as if, after labeling the required variables, the solution does not bind the optimization variable).

6 The entity performing the search is an `IloSolver`. Also, the notion of solution is given by a vector of integers (representing the indexes of the labeled/pruned variables) and a vector of pairs, representing the assigned value or bounds of these variables. This explicit solution entity is built towards the recursive calls of `MyGenerate`, which adds on each call the index of the variable being labeled. Once found the solution, it stores it in DS.

7 After storing a solution in `labW` or `fragW` an `IloGoalFalse` is used, triggering backtracking to continue the breadth exploration.

8 Each call to `MyGenerate` increments the counter value.

4.2.4 Resulting Architecture

The resulting $\mathcal{TOY}(\mathcal{FD})$ architecture supporting the search primitives is presented in Figure 4.8. It contains five different layers:

1. **The $\mathcal{TOY}(\mathcal{FD})$ interpreter**. It allows to submit user commands and goals at the system prompt. In Figure 4.8, the goal proposed in Section 4.1.1 is to be solved. Besides its $\mathcal{FD}$ constraints `domain` and `/=`, there is a `lab` strategy. The user is specifying its own variable and value selection criterion by using the functions

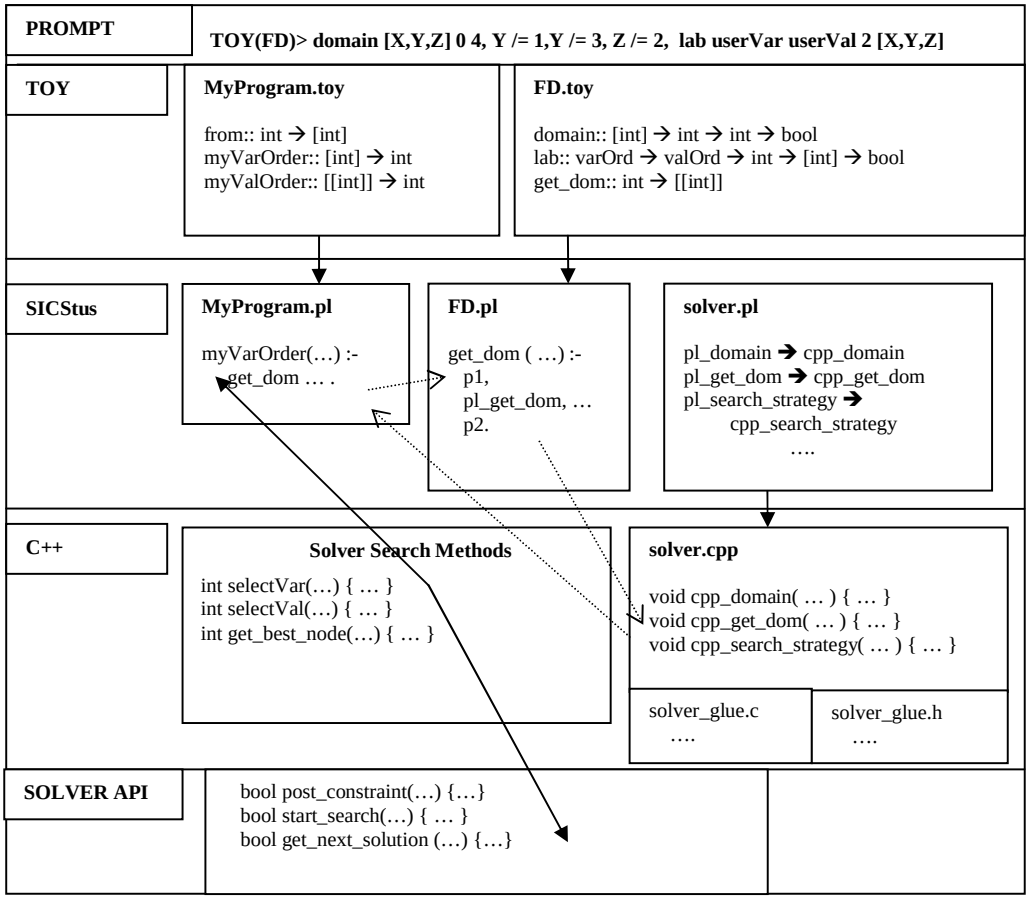

Figure 4.8: Resulting $\mathcal{TOY}(\mathcal{FD})$ Architecture

`myValOrder` and `myVarOrder`, respectively, which rely on auxiliary, primitive and reflection functions (as, for example, `from`, `map` and `get_dom`, respectively).

2. **The $\mathcal{TOY}(\mathcal{FD})$ files defining the language**. They include a file `Prelude.toy` (to specify the prelude functions as, for example, `map`), a devoted file `FD.toy` (specifying the set of $\mathcal{FD}$ constraints supported) and a file `MyProgram.toy` (with the user definitions as, for example, both `from` and `myVarOrder`).

3. **The SICStus implementation of $\mathcal{TOY}(\mathcal{FD})$**. It includes Prolog mate files for all these kinds of functions, implementing all the $\mathcal{TOY}(\mathcal{FD})$ datatypes, functions and operators supported by the system and defined by the user. The file `solver.pl` supports the communication from SICStus to C++ by specifying the set of SICStus predicates S being implemented in C++ functions.

4. **The C++ interface to the solver**. It includes the file `solver.cpp`, containing the set of C++ functions implementing S. Also, it contains the auxiliary files containing the extra C++ functions, data structures and new solver specific classes

extending the library (needed to implement S). This includes the new C++ class `MyGenerate` in Gecode and ILOG Solver (the latter also including `MyInstantiate`). They are devoted to implement the `lab` strategy, and contain methods for the variable and value selection. Figure 4.8 shows how these methods can access either to the solver API (if a predefined criterion is being selected), or (focusing on the variable selection criterion) come back to the SICStus file `MyProgram.pl`, executing the SICStus predicate `myVarOrder` (implementing the user $\mathcal{TOY}(\mathcal{FD})$ function `myVarOrder`). In the example, the latter case holds, and it is shown how the execution traverses the SICStus and C++ layers, as a cycle is performed between the SICStus predicate `myVarOrder`, its auxiliary SICStus predicate `get_dom` (which belong to S) and its C++ implementation in `solver.cpp`.

5. **The C++ API of the solver**. Accessed by the C++ methods interfacing the solver. In the case of Gecode, this layer also includes the solver implementation, as it is open software.

4.3 Performance

This section analyzes the new performance achieved by $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$. The benchmark set of CP($\mathcal{FD}$) problems of Section 3.4 is revisited. For each problem, the structure of its solutions is discussed, and it is pointed out how the use of the proposed search strategies allows to reduce the search exploration to find them. Thus, for each problem, two $\mathcal{TOY}(\mathcal{FD})$ models are created: The one named $problem\_bs.toy$, which was the one used in Section 3.4, and applies a single `labeling` primitive as its search strategy. Another one named $problem\_is.toy$, which is created now for this section, and improves the search strategy of its mate bs model with some of the new proposed search primitives .

All the $\mathcal{TOY}(\mathcal{FD})$ models are available at: `http://gpd.sip.ucm.es/ncasti/models.zip`. For the sake of simplicity, from now on the different versions of the models will be simply referred to as bs and is.

4.3.1 Analyzing the Applied Search Strategies

Magic Sequence. The bs model used a single `labeling [ff] L` as its search strategy. Analyzing the solutions of the problem it is observed that, if the parameter $n \geq 9$ then the sequence follows the pattern: $L \equiv [(n-4), 2, 1, 0, 0, \ldots, 1, 0, 0, 0]$. The new search strategy of the is model first applies `labB unassignedRightVar smallestVal 3 L`, `labB unassignedRightVar largestVal 1 L`, which matches the last four variable $1, 0, 0, 0$ pattern. At that point, propagation leads to $L \equiv [(n-4), A, B, C, 0, \ldots, 1, 0,$

$0, 0]$ (with A in $1..3$, B in $0..1$ and C in $0..1$), highly reducing the search space the further `labeling` has to deal with.

Queens. The bs model used a single `labeling [ff] L` as its search strategy. Analyzing the solutions of the problem, an intuitive way for reducing the initial search space of the problem consists of: Splitting the n variables into k variable sets $(vs_1, vs_2, \ldots, vs_k)$ (where consecutive variables are placed in different variable sets). Splitting the initial domain $1 \ldots n$ into k different intervals $(1..(n/k), \ldots, (n/k) * (i-1) + 1..(n/k) * i, \ldots, (n/k) * (k-1) + 1..n)$. Assigning the variables of vs_i to the i^{th} interval.

Thus, the new search strategy of the is model first applies `split_into_3 L ([], [], []) == (K1,K2,K3), fragB (partition 3) unassignedLeftVar firstRight 0 K1, fragB (partition 3) unassigned- LeftVar firstMiddle 0 K2, fragB (partition 3) unassignedLeftVar first- Left 0 K3`. This splits into three sets the variables and their domains, highly reducing the search space the further `labeling` has to deal with.

Langford's Number. The bs model used a single `labeling [ff] L` as its search strategy. Analyzing the solutions of the instances proposed, it is observed that they follow the pattern: $L \equiv [X1, X2, \ldots, A, B, C, D, E, F]$, with an inductive mapping between the set of variables {A,B,E,F} and the set of values {1,2,3,4}. Thus, the new search strategy of the is model first applies `fragB (partition ((round ((M*N)/4)) - 1)) unassigned- RightVar firstLeft 0 [A,B,E,F], labW unassigned- RightVar smallestTotalDomain 0 [A,B,E,F]`.

The `fragB` fragments the domain of `[A,B,E,F]` in the `(M*N)/4` intervals of values `1..4`, `5..8`, ..., `M*N-3..M*N`. It selects the first interval starting from the left (i.e., the smallest one), and it precludes any further backtracking to explore the remaining intervals. Then, with the domain of `[A,B,E,F]` pruned to be in `1..4`, `labW` labels them, exploring all their feasible combinations before selecting the one leading to the smallest search space for `L`. Thus, it is clear that the use of the previous `fragB` is crucial for the success of the `labW` strategy (although the use of the proper `fragB` might cause a loss of completeness for other instances of the problem). A deep breath exploration with `labW` supposes a tradeoff between obtaining an ordered hierarchy of relevant intermediate tree-level nodes and the computational effort to obtain this hierarchy. With an initial domain of `1..M*N`, the feasible combinations of values for `[A,B,E,F]` is unaffordable in terms of time and memory. However, with a domain of `1..4` (and knowing that they are constrained with an $all\_different$) the amount of feasible combinations is reduced to, at most, 24 (which is clearly affordable).

Golomb Rulers. The bs model used a single `labeling [toMinimize Mn] M` as its search strategy. Analyzing the solutions of the instances proposed, it is observed that the initial domain of their variables is huge, and that the value they take in the optimal solution is not far away from their initial lower bound. For example, in G-11 (for which $M \equiv [0, A, B, ..., H, I, J]$), the initial domain of the last three variables is H in $36..1020$, I in $45..1021$ and J in $55..1023$ (with known optimal solution 64, 70 and 72,

respectively) and the initial domain of the first three variables is 0, A in $1..977$ and B in $3..987$ (with know optimal solution 0, 1 and 4, respectively).

Thus, an intuitive way of reducing the initial search space is by reducing as much as possible the upper bound of these variables. The new search strategy of the is model first applies `fragW (partition 3) unassignedRightVar smallestSearchTree 2 L`, `fragW (partition 12) unassignedLeftVar largestSearchTree 2 L`, fragmenting first the last two variables and then the first two. Note that, whereas the former selects as *best* intermediate node the one minimizing the remaining search space, the latter selects the one maximizing it (which intuitively makes sense, as the smaller interval is the one pruning the least the upper bound of the first two variables, thus pruning the least the rest of the variables).

4.3.2 Running the Experiments

Table 4.2 revisits the experiments of Section 3.4, comparing the performance of mate bs and is instances. Columns `bs` and `is` represent the CPU solving time of bs and is (respectively), both of them using incremental propagation mode. Column `Sp-Up_bs` represents the speed-up (of the CPU times) of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ for running the bs models. Similarly, column `Sp-Up_is` represents the speed-up (of the CPU times) of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ for running the is models. Finally, column `off/on` focuses on each concrete $\mathcal{TOY}(\mathcal{FD})$ version, representing the speed-up of is w.r.t. bs.

The results allow to draw the following conclusions:

- First, the use of the new search strategies is encouraging, as the performance of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ for solving is instances is better than the achieved for solving bs ones (besides Q-90 and L-119, where $\mathcal{TOY}(\mathcal{FD}i)$ spends about 0.4 seconds more in solving is). In any case, the differences range from 1.05 times faster to more than 1000 times, so a more detailed analysis by problems and instances is required.

For Queens and Langford's is instances, the better performance achieved by $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ clearly scales as the sizes of the instances scale. More specifically, for Q-90 and L-119 (solved in tenths of seconds) $\mathcal{TOY}(\mathcal{FD}g)$ achieves an improvement of 1.28 times and 1.05, respectively. This improvement grows an order of magnitude for Q-105 and L-127 (with an improvement of 12.50) and two orders of magnitude for Q-120 and L-131 (with an improvement of 1443.11 and 298.58, respectively). In $\mathcal{TOY}(\mathcal{FD}i)$, it is observed the same growing pattern, but it is less noticeable. For Q-90 and L-119 the is performance is even worse than the bs one. Then, for Q-105 and L-127 the is performance improves a 1.61 and a 3.70, respectively, but still in the same order of magnitude as bs. Finally, for Q-120 and L-131 the is performance reaches the two orders of magnitude improvement w.r.t. bs (138.74 and 73.11, respectively).

For Magic is instances, the better performance achieved by $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ remains stable as the size of the instances scale (with around a 1.49-1.52 for $\mathcal{TOY}(\mathcal{FD}g)$ and a 1.30-1.32 for $\mathcal{TOY}(\mathcal{FD}i)$). Last, for Golomb is instances the

Instance	bs	Sp-Up_bs	is	Sp-Up_is	off/on
M-400 FDi	0.530	1.00	0.402	1.00	1.32
M-400 FDg	0.422	1.25	0.280	1.43	1.52
M-900 FDi	2.53	1.00	1.95	1.00	1.30
M-900 FDg	2.00	1.27	1.34	1.45	1.49
Q-90 FDi	0.110	1.00	0.514	1.00	0.21
Q-90 FDg	0.078	1.41	0.061	8.33	1.28
Q-105 FDi	1.25	1.00	1.28	1.00	1.61
Q-105 FDg	1.05	1.19	0.08	10.00	12.50
Q-120 FDi	154.00	1.00	1.11	1.00	138.74
Q-120 FDg	129.88	1.19	0.09	12.50	1443.11
L-119 FDi	0.530	1.00	0.984	1.00	0.54
L-119 FDg	0.296	1.79	0.282	3.45	1.05
L-127 FDi	4.35	1.00	1.17	1.00	3.70
L-127 FDg	4.62	0.94	0.39	3.03	12.50
L-131 FDi	87.00	1.00	1.19	1.00	73.11
L-131 FDg	98.53	0.88	0.33	3.57	298.58
G-9 FDi	0.421	1.00	0.109	1.00	3.85
G-9 FDg	0.250	1.69	0.062	1.75	4.00
G-10 FDi	3.56	1.00	1.47	1.00	2.44
G-10 FDg	2.11	1.69	0.84	1.75	2.50
G-11 FDi	72.65	1.00	43.98	1.00	1.64
G-11 FDg	42.01	1.72	24.85	1.75	1.69

Table 4.2: Performance of $\mathcal{TOY}(\mathcal{FD})$ using the Search Strategies

better performance achieved by $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ decreases around as the instances scale up, with a 4.00, 2.50 and 1.69 improvement of $\mathcal{TOY}(\mathcal{FD}g)$ for G-9, G-10 and G-11, respectively, and a 3.85, 2.44 and 1.64 of $\mathcal{TOY}(\mathcal{FD}i)$.

- Second, it is clearly observed that the improvement achieved by $\mathcal{TOY}(\mathcal{FD}g)$ for is instances is greater than the one achieved by $\mathcal{TOY}(\mathcal{FD}i)$, revealing that the approach Gecode offers to extend the library with new search strategies is more efficient than the one of ILOG Solver. That is, for any is instance, the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ is greater than the achieved for its mate bs instance.

For each problem, the impact of is obviously depends on how well its improved search strategy captures the structure of the problem solutions. Two different behaviors are observed: First, for Queens and Langford's is instances the speed-up improvement achieved w.r.t. bs instances increases as the instances scale up: An increasing from 1.41 to 8.33, from 1.19 to 10.00 and from 1.19 to 12.50 for Q-90, Q-105 and Q-120, respectively. An increasing from 1.79 to 3.45, from 0.94 to 3.03 and from 0.88 to 3.57

for L-119, L-127 and L-131, respectively. Second, for Magic and Golomb is instances the speed-up improvement achieved w.r.t. bs instances remains stable as the instances scale up: From 1.25 to 1.43 and from 1.27 to 1.45 for M-400 and M-900, respectively. From 1.69 to 1.75 for both G-9 and G-10, and from 1.72 to 1.75 for G-11.

4.4 Related Work

Regarding the search capabilities provided by other multi-paradigm programming systems, it should be pointed out that PAKCS and Mozart respectively provide the primitives `labeling` and `distribute`, implementing a depth first search exploration of the tree. They perform domain labeling or domain splitting on each node (`var = val \/ var != val` or `var <= val \/ var > val`, respectively). For variable selection, they provide first unbound and first fail. For value selection, they provide `min`, `middle` and `max` values.

Thus, the search capabilities of $\mathcal{TOY}(\mathcal{FD})$ offer appealing possibilities for CFLP($\mathcal{FD}$). First, all the functionality of PAKCS and Mozart is also achieved in $\mathcal{TOY}(\mathcal{FD})$ by using its `labeling` primitive. Besides that, the new primitives presented in Section 4.1 for $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ provide novel concepts for search, as performing an exhaustive breadth exploration of the search tree, further sorting the satisfiable solutions by a specified criterion, fragmenting the variables pruning each one to a subset of its domain values instead of binding it to a single value, and applying the labeling or fragment strategy only to a subset of the variables involved. Finally, regarding variable and value selection, a wide range of criterion is offered, which includes specifying a user-criterion directly via $\mathcal{TOY}(\mathcal{FD})$ functions.

The approach of taking advantage of both the high expressivity of $\mathcal{TOY}(\mathcal{FD})$ (for specifying search strategies) and of the high efficiency of Gecode and ILOG Solver (to accomplish them) can be related to the one followed in *Search Combinators* [170]. It provides a lightweight and solver-independent method bridging the gap between a conceptually simple search language (high level, functional and naturally compositional) and an efficient implementation (low-level, imperative and highly non-modular). It allows to define application tailored strategies from a small set of primitives, resulting in a rich search language for the user and a low implementation cost for the developer of a constraint solver.

The search language of $\mathcal{TOY}(\mathcal{FD})$ is more rigid than the one of [170], but some of the features provided by the search combinators can be matched with the new set of primitives presented in Section 4.1: The basic primitive heuristics `base_search` and `prune` can be obtained with the primitive `lab`, controlling the exact number of variables to be labeled (which allows $\mathcal{TOY}(\mathcal{FD})$ to support composite search strategies). Regarding the set of combinators proposed, $\mathcal{TOY}(\mathcal{FD})$ matches {`let`, `assign`, `post`} via intermixing search procedures with constraint posting, and {`and`, `or`} via the com-

posed search strategies presented in Section 4.1.3. Finally, the $\mathcal{TOY}(\mathcal{FD})$ primitives are also extensible, as users can program their own criterion at $\mathcal{TOY}(\mathcal{FD})$ level with no extra implementation effort at the SICStus and C++ layers of the architecture (as it can be seen in Figure 4.8).

Similar approaches to search combinators are proposed for FP, with Monadic Constraint Programming, and for LP, with the library TOR [171] (available in SWI-Prolog). They decouple the notion of defining the search tree from the notion of defining the search method. In $\mathcal{TOY}(\mathcal{FD})$, it is not possible to specify the way to explore the search tree (as, for example, by limited discrepancy search). However, the primitives `labW` and `fragW` perform a breadth search exploration. Moreover, at least in $\mathcal{TOY}(\mathcal{FD}g)$ it would not be difficult to support new ways of exploring the search tree, as the Gecode library provide the mechanisms to implement them.

4.5 Conclusions

The combinatorial nature of the CSP's and COP's tackled with $\mathcal{TOY}(\mathcal{FD})$ turn the search exploration into a key factor for the performance of the system. The analysis of chapter 3 has revealed that, as long as the instances of these problems scale up enough, the percentage of CPU time devoted to search exploration tends to converge to almost the 100%. Thus, the performance of $\mathcal{TOY}(\mathcal{FD})$ becomes purely CP($\mathcal{FD}$) dependent, i.e., its CPU time directly comes from the performance of its concrete solver in achieving the pure CP($\mathcal{FD}$) mechanism of performing a search exploration by propagating basic and global constraints. This leads to two different approaches to improve the $\mathcal{TOY}(\mathcal{FD})$ performance: A first one, focusing on the solver accomplishing the search, replacing the current $\mathcal{FD}$ solver of the system by new state-of-the-art ones. That is, keeping the same $\mathcal{TOY}(\mathcal{FD})$ model for a problem (specifically, keeping the same $\mathcal{FD}$ constraint network and search strategy), rely on new solvers capable of solving it faster. A second one focusing on the search to be accomplished, replacing it by an *ad hoc* one which exploits the knowledge about the structure of the problem and its solutions. That is, keeping the same solver to accomplish the search, modify the $\mathcal{TOY}(\mathcal{FD})$ model to specify a search strategy requiring less search exploration to find the solutions.

Chapter 3 has followed the first approach, interfacing the state-of-the-art C++ CP($\mathcal{FD}$) solvers of Gecode and ILOG Solver into $\mathcal{TOY}(\mathcal{FD})$, giving rise to the new versions $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$, respectively. Departing from these system versions, this chapter has followed the second option, enhancing the $\mathcal{TOY}(\mathcal{FD})$ language with new search primitives allowing a better specification of *ad hoc* search strategies. The motivation of this approach has been to take advantage of both the high expressivity of $\mathcal{TOY}(\mathcal{FD})$ (for specifying search strategies), and of the high efficiency of Gecode and ILOG Solver (for accomplishing them). The techniques used can be easily adapted to other CLP($\mathcal{FD}$) or CFLP($\mathcal{FD}$) systems implemented in Prolog and interfacing external

C++ CP($\mathcal{FD}$) solvers.

The chapter has presented eight new $\mathcal{TOY}(\mathcal{FD})$ search primitives, describing their declarative semantics from a solver independent point of view, and using examples to show their application. It has emphasized the novel concepts those primitives include, as performing an exhaustive breadth exploration of the search tree further sorting the satisfiable solutions by a specified criterion, fragmenting the variables pruning each one to a subset of its domain values instead of binding it to a single value, and applying the labeling or fragment strategy only to a subset of the variables involved. It has also pointed out how expressive, easy and flexible it is to specify some search criterion at $\mathcal{TOY}(\mathcal{FD})$ level, as well as how easy is to use model reasoning to apply different search strategies (setting different search scenarios) to the solving of a CP($\mathcal{FD}$) problem.

A new version of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ including these search primitives has been presented. It has been described their implementation in Gecode and ILOG Solver, by extending their libraries relying on their underlying search mechanisms. It has been observed that these search mechanisms are quite different in Gecode (`Search Engine`, `Brancher` methods, hybrid recomputation) and ILOG Solver (`IloGoal`, goal constructor, goal stack). Thus, an abstract view of the requirements needed to integrate the search strategies in $\mathcal{TOY}(\mathcal{FD})$ has been first presented (with the scheme further instantiated to Gecode and ILOG Solver). Also, the impact of the implementation of the search primitives on the system architecture has been analyzed, revealing that the specification of search criterion at $\mathcal{TOY}(\mathcal{FD})$ level has an inherent overhead due to the recursive interaction between the SICStus Prolog and C++ layers.

Finally, the benchmark of Chapter 3 has been revisited, discussing the structure of the solutions of each problem, and pointing out how the use of the proposed search strategies allow to reduce the search exploration to find them. Mate $\mathcal{TOY}(\mathcal{FD})$ models with an improved *ad hoc* search strategy have been developed. It has been shown that the use of the new search strategies improve the performance of both $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$, but the improvement achieved (ranging in 1.05 times faster to more than 1000 times) is dependent on the concrete problem and instance solved: Whereas for Queens and Langford's instances the better performance achieved clearly scales as the sizes of the instances scale, for Magic ones it remains stable, and for Golomb ones it decreases. Moreover, the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ is greater for the new improved $\mathcal{TOY}(\mathcal{FD})$ models, revealing that the approach Gecode offers to extend the library with new search strategies is more efficient than the one of ILOG Solver.

Part III

Real-Life Applications of $\mathcal{TOY}(\mathcal{FD})$

The second research part of the thesis is focused in describing real-life applications of $\mathcal{TOY}(\mathcal{FD})$. Chapter 5 describes the modeling and solving of an Employee Timetabling Problem coming from the communication industry. Chapter 6 describes the application of $\mathcal{TOY}(\mathcal{FD}g)$ to an empirical analysis process (based on real-life benchmarks) to test the hardness of the Bin Packing Problem.

Chapter 5

An Employee Timetabling Problem

The use of the classical CP($\mathcal{FD}$) benchmarks in Chapters 3 and 4 has shown that interfacing external C++ CP($\mathcal{FD}$) solvers and applying *ad hoc* search strategies have improved the solving performance of $\mathcal{TOY}(\mathcal{FD})$. However, the modeling of these problems is purely CP($\mathcal{FD}$) dependent and, moreover, quite naïve. That is, there are no extra difficulties for modeling the problems but the ones for expressing their associated $\mathcal{FD}$ constraint networks, and the formulation of these constraint networks can be considered easy enough (in the sense that they do not require a high expressivity). Thus, it is claimed that the classical CP($\mathcal{FD}$) benchmarks do not exploit the capabilities of a general-purpose language as $\mathcal{TOY}(\mathcal{FD})$, with a high expressivity coming from the functional and logic features the language provides.

The main contribution of this chapter is to apply $\mathcal{TOY}(\mathcal{FD})$ to a real-life Employee Timetabling Problem (from now on, just ETP), exploiting both the high expressivity of the language and the higher solving performance achieved with the techniques previously described in Chapters 3 and 4. An ETP is concerned with assigning a number of employees to a given set of shifts over a fixed period of time, taking into account the employees qualifications, constraints and preferences. Probably the most paradigmatic example of an ETP is the Nurse Rostering Problem (NRP), which has been extensively studied in the last decades [38].

The proposed ETP comes from the communication industry, and it can be seen as a particular case of the NRP. A first formulation of the problem was proposed in [153]. On it, a partial solution was constructed (based on the knowledge about a complete solution structure) and was used as a seed for triggering the search looking for feasible complete solutions. Then, the problem was reformulated from scratch in [47], with the aim to tackle a concrete instance (with a fixed structure), which included a new solving approach (based on problem decomposition) as well as new requirements

(also embodying optimization). Maintaining both the formulation of [47] (which is non-monolithic and includes CP($\mathcal{FD}$) independent components) and its solving approach, this chapter presents a new parameterizable version of the ETP, now applicable to a wide set of different instances.

The chapter is organized as follows: Section 5.1 presents a description of the problem and Section 5.2 describes the solving approach to tackle it. Section 5.3 presents a parametric algorithm implementing the solving approach, which is the one being followed by $\mathcal{TOY}(\mathcal{FD})$ to model the problem. Section 5.4 presents the results of running different instances of the problem in the three $\mathcal{TOY}(\mathcal{FD})$ versions, and compares these results with the ones obtained for solving classical CP($\mathcal{FD}$) benchmarks. Section 5.5 presents some related work. Finally, Section 5.6 reports conclusions.

5.1 Problem Description

A department is filling its employee timetable for the next nd days. The timetabling contains different kinds of working days $ws \equiv \{ws_1, \ldots, ws_k\}$, each one with an associated amount of shifts $ws_i \equiv [s_{wsi1}, \ldots, s_{wsim}]$. The days $dc \equiv [dc_1, \ldots, dc_{nd}]$ of the timetabling are identified by the kind of working day they are (i.e., each dc_i takes a value among $\{ws_1, \ldots, ws_k\}$). If ws_i provides m shifts, then for each day $dc_x = ws_i$ there must be m available workers at the department (each shift is assigned to a single worker). The department employs w workers. $nt * ntw$ of them are regular workers divided into nt teams of ntw workers: $\{w_1, \ldots, w_{ntw}\}$ belong to team t_1, $\{w_{ntw+1}, \ldots, w_{2*ntw}\}$ belong to team t_2, and so on. Note that, obviously, this also includes the possibility of having just one team containing all the regular workers. In any case, there is an extra worker ew, which belongs to no team and is only selected by demand for coping with regular workers absences. Whereas ew must be available for the nd days, regular workers must be absent some days, which is provided (in advance) by $abs \equiv \{(w_{i1}, d_{j1}), \ldots, (w_{il}, d_{jl})\}$, of pairs (regular worker, day).

Each team t_i is selected once every nt days. If t_i is selected on day dc_j only $\{w_{(i-1)*ntw+1}, \ldots, w_{i*ntw}\}$ (the regular workers of that team) and ew are able to work at the department on that day. The extra worker ew can be selected to work just one out of each er consecutive days. Note that, obviously, this also includes the possibility of working every day, by simply setting er to one. For each day dc_j, the selected t_i provides (due to absences) $0 \leq a \leq ntw$ available workers to be assigned to the m available shifts of the day. This gives rise to up to three different situations: First, if $a \geq m$ then ew and any possible remaining regular worker do not have to work (they are assigned to shifts of 0 hours). Second, if $a = m - 1$ then ew is selected to work, coping with one of the m shifts of dc_j. Third, if $a < m - 1$ then there is no possible assignment, and t_i cannot be selected for dc_j.

Given $s \equiv \{s_1, \ldots, s_g\}$ as the set of different shift types that ws provides, T_{t_i,s_z} is a

measure of the distribution of shifts of type s_z to the regular workers of t_i. Supposing that each worker of the team w_i is assigned to a certain amount cv_i of shifts of type s_z during the nd days, then T_{t_i,s_z} represents the difference between the maximum and minimum of these cv values. Shift distribution is constrained by T, representing the maximum value any $T_{t,s}$ can take. Note that this constraint can be easily disregarded by setting T to $(nd/nt)+1$ or higher: Team t_i works each nt days, so it will work (nd/nt) or $(nd/nt)+1$ days during the timetable (where / stands for the integer division). As each worker is assigned to a single shift each of those days, the highest possible T_{t_i,s_z} would be the one in which one worker of t_i is only assigned to shifts of kind s_z, whereas another worker of the team is assigned to none of them.

Scheduled timetabling contains h working hours, so each regular worker is expected to work $h/(nt * ntw)$ hours. Any more worked hour is considered as an extra hour. Optimization arises in the problem because the department must pay regular workers for each extra hour they work, and any hour that ew works is paid as ef extra hours of a regular worker. Note that ew can be treated as a regular worker by setting ef to one. An optimal schedule minimizes the extra hour payment. It is important to point out that T does not belong to the optimization function. The objective is to minimize the extra hour payment, not T. However, T represents a measurement of the fairness of the scheduled timetabling, and it is parameterizable by the user. For example, two different schedules implying 16 extra hours (one assigning them to a worker and the other one dividing them among the workers of a team) can be equivalent regarding optimality, but the second is *fairer*.

Instance Example: A timetabling is scheduled for next week, which starts on Monday. Whereas each working day contains three shifts to be accomplished (of twenty, twenty two and twenty four hours, respectively), each weekend day contains just two (both of them of twenty four hours). The department employs thirteen workers: Twelve of them regular workers, divided into three teams of four workers. Some of these regular workers are not available all days, with their absences described by $abs = [(w_1, d_1), (w_2, d_1), (w_5, d_1), (w_5, d_6), (w_6, d_1), (w_6, d_6), (w_7, d_1), (w_7, d_6), (w_{10}, d_1), (w_{10}, d_6), (w_{11}, d_1), (w_{11}, d_6), (w_{12}, d_1), (w_{12}, d_6)]$. The department also employs an extra worker, which can be selected to work at most one out of each three consecutive days, and whose extra hours are paid the double. Finally, small deviations (of one per kind of shift) are allowed between the different workers of a team. Table 5.1 summarizes the list of parameters used in the problem description. Whereas columns `Variable Name` and `Variable Description` represent the variable name and its informal description (respectively), column `Example` represents the value of the variable for the concrete instance presented.

Variable Name	Variable Description	Example
nd	Number of days	$nd = 7$
dc	Day classification	$dc = [1, 1, 1, 1, 1, 2, 2]$
ws	Different kind of working days	$ws = [[20, 22, 24], [24, 24]]$
w	Number of workers	$w = 13$
nt	Number of teams	$nt = 3$
ntw	Number of team workers	$ntw = 4$
$w_{(i-1)*ntw+1}$, ..., w_{i*ntw}	Workers of team t_i	t_1; t_2; $t_3 = \{w_9, w_{10}, w_{11}, w_{12}\}$;
ew	Extra worker	$ew = w_{13}$
er	Period of consecutive days on which ew can work at most one of them	$er = 3$
abs	Absences of (regular workers, days)	$abs = [(1, 1), (2, 1), \ldots, (12, 6)]$
s	Different kind of shifts	$s = [0, 20, 22, 24]$
T_{t_i,s_z}	Measure of the distribution of shifts of kind s_j to the regular workers of team t_i	$T_{t_1,s_0} = 1$
$cv_1, \ldots, cv_{ntw}$	Number of shifts of kind s = 0 the regular workers of t_1 are assigned to	$cv_1 = 2$; $cv_2 = 1$; $cv_3 = 1$; $cv_4 = 1$
T	Maximum value for the Tt_i, s_j	$T = 1$
ef	Factor for the working hours of ew	$ef = 2$

Table 5.1: List of Variables used in the Problem

5.2 Solving Approach

The proposed description abstracts the problem as an entity receiving nd, nt, ntw, er, ef, ws, abs, dc and T as its input parameters, and computing the pair $(timetabling, eh)$ as a result. $timetabling$ is an $w \times nd$ assignment that can be represented by a matrix, where each position (i, j) represents the shift assigned on day j to worker i (e.g., in the instance example, $timetabling$ is a 13×7 matrix). eh represents the total amount of extra hours of this assignment.

An intuitive way to model the problem is to use $timetabling$ as the set of $\mathcal{FD}$ variables, with dc and ws being used to determine the initial domain of the variables of each kind of day. However, the problem requirements requesting only one team to work each day, and each team to be selected each nt days produces strong dependencies between the $timetable$ variables: As soon as a regular worker w_k (belonging to team t_i) is selected for working on day d_j (by assigning it to a working shift $s_{jk} > 0$) then it is known that t_i is going to be selected to work as well for days d_j, d_{j+nt}, $d_{j+(2*nt)}$, and so on. This precludes the other teams from working on these days, as well as precludes t_i from working on days d_{j+1}, $d_{j+(nt-1)}$, $d_{j+(nt+1)}$, $d_{j+(2*nt-1)}$, and so on. Figure 5.1 presents these dependencies for the instance example, showing the implications of assigning $timetabling_{1,2} = 20$. Thus, although these dependencies are perfectly modeled (for example, by using propositional implication constraints), a more efficient approach to model the problem is to use $Table$ instead of $timetabling$. $Table$ is an $(ntw + 1) \times nd$ matrix where, for each d_j, the first ntw rows represent the regular workers of the team t_i selected for working on d_j. The row $(ntw + 1)$ represents ew. To use this approach, bidirectional mappings between $timetabling$ and $Table$ are needed. They require an additional register tda (teams to days assignment), indicating which team works each day. Figure 5.2 presents such mapping for the instance example, where the first four rows refer to {w_1, w_2, w_3, w_4} on days 1, 4 and 7, to {w_9, w_{10}, w_{11}, w_{12}} on days 2 and 5, and to {w_5, w_6, w_7, w_8} on days 3 and 6.

As it can be seen, a single $Table$ does not explore the whole search space of $timetabling$, but only the subset associated to the concrete tda selected. Thus, the solution found in $Table$ can only be considered suboptimal, in the sense that it is only optimal w.r.t. this $timetabling$ search space subset. In fact, the nt teams lead to up to $nt!$ possible $tda \equiv \{tda_1, \ldots, tda_{nt!}\}$ (each of them with its associated $Table$ to be explored). To find the optimal solution, not all the different $Table$ must be explored, but only those ones associated to feasible tda. More specifically, a tda_i is said to be satisfiable if, for each day, there are enough workers of the selected team to accomplish the scheduled shifts, and ew is never selected to work more than one out of each er consecutive days. Finding an unfeasible tda_i allows to save the exploration of $1/nt!$ of the search space of $timetabling$. In the instance example, only two of the six tda are feasible, as due to the absences provided by abs, only team t_1 can work on d_1. The two feasible assignments are then assigning t_2 (respectively t_3) to d_2 and t_3 (respectively t_2) to d_3.

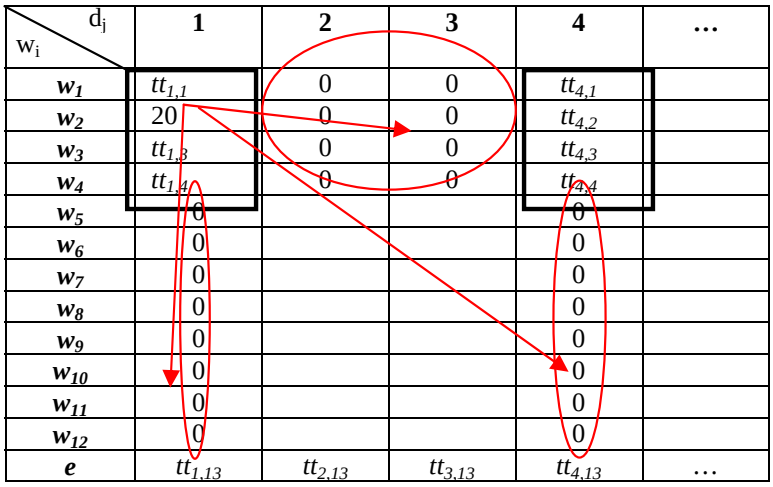

Figure 5.1: Team Dependencies in $timetable$

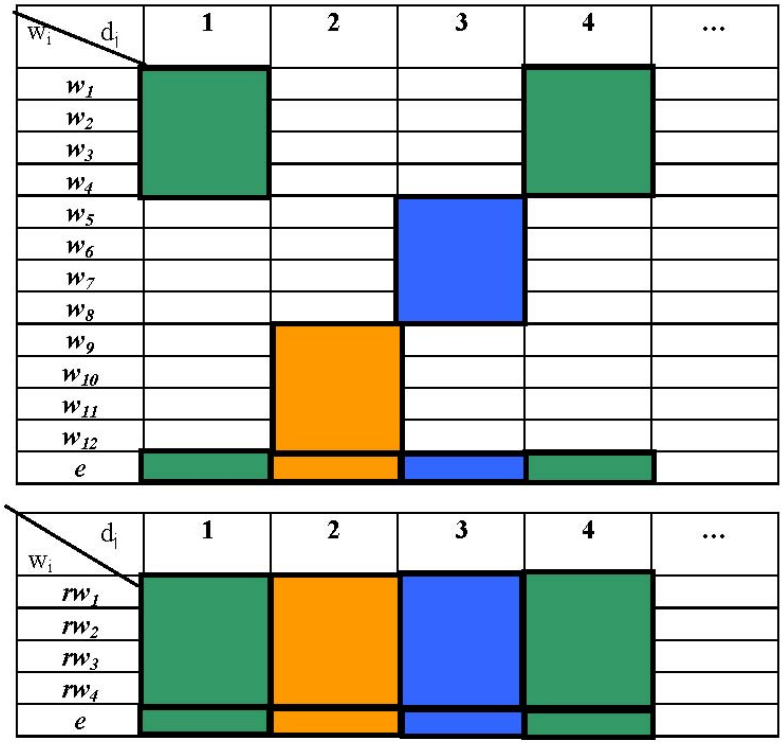

Figure 5.2: Mapping from $timetable$ to $Table$

Last but not least, the nt teams are initially linked by the fact that, if needed, ew can only be selected to work one out of each er consecutive days. However, as a feasible tda entails the resting constraint of ew, its associated $Table$ can be split into nt independent subproblems (exponentially easier to be solved). In the instance example, the associated $Table$ of Figure 5.2 is split into tt_1 (with columns 1, 4 and 7), tt_2 (with columns 2 and 5) and tt_3 (with columns 3 and 6).

In summary, the natural modeling based on $timetabling$ requires $w \times nd$ variables (in the instance example it is $13 \times 7 \equiv 91$). Supposing that, due to the requested shifts, the initial domain of each variable was {0, 20, 22, 24}, then the initial search space would contain $4^{91} \equiv 6,12 * 10^{54}$ candidates. Thus, the proposed solving approach based on tda improves the solving efficiency by:

- Using a $Table$ of $(ntw+1) \times nd$ variables, saving $ntw * (nt-1) \times nd$ variables w.r.t. the $timetabling$ approach. In the instance example, $Table$ is 5×7, saving a 61% of the $timetabling$ variables, and thus reducing the search space to $4^{0.39*91} \equiv 2.32 * 10^{21}$ (cf. Figure 5.2).
- Splitting $Table$ into nt independent subproblems, exponentially easier to be solved. In the instance example, there are three independent teams (cf. Figure 5.2, columns green, orange and blue, respectively), and thus the search space of each of them is $4^{0.13*91} \equiv 1.32 * 10^7$. As the three subproblems must be solved, then the search space being explored turns into $3,96 * 10^7$.
- Exploring only any $Table_i$ associated to a feasible tda_i. In the instance example, only two tda of the $3! = 6$ possible ones are feasible. Thus, to find the optimal solution the search space being explored turns into $7.92 * 10^7$, which is much smaller than the original $6,12 * 10^{54}$ one.

5.3 Algorithm

In this section, a *parametric timetabling* p_tt algorithm implementing the solving approach of Section 5.2 is presented. It computes an optimal assignment for $timetabling$ by: Finding any feasible tda_i, mapping it to $Table_i$, splitting $Table_i$ into nt independent subproblems tt_{i1}, ..., tt_{int}, solving them sequentially, and mapping the suboptimal computed $Table_i$ to $(timetabling_i, eh_i)$.

Thus, the algorithm relies on a four stage process: $team\_assign$, tt_split, tt_solve and tt_map. The stage $team\_assign$ just concerns with finding any feasible bijection tda_i and creating its associated $Table_i$. Starting from a feasible bijection tda_i, the stages tt_split, tt_solve and tt_map are executed, splitting $Table$ into $tt_1, \ldots, tt_{nt}$, solving the different tt_i sequentially and mapping the computed suboptimal $Table_i$ to $(timetabling_i, eh_i)$, respectively. Then, backtracking is triggered to $team\_assign$, finding a new tda_j

and executing again the stages tt_split, tt_solve and tt_map to obtain a new suboptimal $(timetabling_j,\ eh_j)$. Finally, when backtracking to $team\_assign$ produces no more feasible tda, the computed ($timetabling_1,\ eh_1$), ..., ($timetabling_k,\ eh_k$) are compared, outputting as a result the one with minimum eh.

The call $p\_tt\ nd\ nt\ ntw\ er\ ef\ ws\ abs\ dc\ T\ P\ W\ SS = (timetabling,\ eh)$ runs the algorithm. The special input parameters P, W and SS are provided to configure the $\mathcal{FD}$ constraint solver. Once again, P sets the propagation mode to *incremental* or *batch*. W and SS set the search strategy to order the variables by workers or days, and label the variables by *first_unbound* or *first_fail*, respectively. Focusing on the instance example, and setting the solver to use incremental propagation mode and label the variables by workers in a textual order, the call to p_tt results to be:

p_tt 7 3 4 3 2 [[20, 22, 24], [24, 24]] [(1, 1), (2, 1), (5, 1), (5, 6), (6, 1), (6, 6), (7, 1), (7, 6), (10, 1), (10, 6), (11, 1), (11, 6), (12, 1), (12, 6)] [1, 1, 1, 1, 1, 2, 2] 1 $true\ workers\ fst\_unb =$

([[0, 0, 22, 24, 0, 0, 0, 0, 0, 0, 0, 0, 20],
[0, 0, 0, 0, 0, 20, 22, 24, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 22, 24, 0],
[0, 20, 24, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 24, 22, 20, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0, 24, 0, 0, 0, 24],
[24, 24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]], 133)

Next, p_tt is described in detail. The different stages are presented separately and, for each of them, the steps being followed are enumerated. Each step is univocally identified as "**p_tt_stagename_stagestep**" (where $team\_assign$ is named ta, tt_split is named sp, tt_solve is named so and tt_map is named ma). The execution of the instance example is used to guide this p_tt description. In particular, it is pointed out the computation of any data structure, as well as the posting of any $\mathcal{FD}$ constraint to the solver.

5.3.1 Stage team_assign:

The call $team\_assign\ nd\ nt\ ntw\ er\ ws\ abs\ dc\ P = (d, e, oabs, Table)$ runs the stage. In the instance example, this call results to be:

```
team_assign 7 3 4 3 [ [20, 22, 24], [24, 24] ] [ (1, 1), (2, 1),
  (5, 1), (5, 6), (6, 1), (6, 6), (7, 1), (7, 6), (10, 1), (10,
  6), (11, 1), (11, 6), (12, 1), (12, 6), [1, 1, 1, 1, 1, 2, 2]
  true = (d, e, oabs, Table)
```

Two feasible tda_1 and tda_2 are obtained:

```
sol1:
d = [1, 2, 3, 1, 2, 3, 1];
```

```
e = [1, 0, 0, 0, 0, 1, 0];
oabs = [ [1,2,5,6,7,10,11,12], [], [],
         [], [], [5,6,7,10,11,12], [] ];
Table = [ [v1, v2, v3, v4, v5], [v6, v7, v8, v9, v10], [v11, v12,
  v13, v14, v15], [v16, v17, v18, v19, v20], [v21, v22, v23, v24,
  v25], [v26, v27, v28, v29, v30], [v31, v32, v33, v34, v35] ];

sol2:
d = [1, 3, 2, 1, 3, 2, 1];
e = [1, 0, 0, 0, 0, 1, 0];
oabs = [ [1,2,5,6,7,10,11,12], [], [],
         [], [], [5,6,7,10,11,12], [] ];
Table = [ [v1, v2, v3, v4, v5], [v6, v7, v8, v9, v10], [v11, v12,
  v13, v14, v15], [v16, v17, v18, v19, v20], [v21, v22, v23, v24,
  v25], [v26, v27, v28, v29, v30], [v31, v32, v33, v34, v35] ];
```

The following steps are performed to compute the solution:

p_tt_ta_1. $Table$, an $\{(ntw + 1) \times nd\}$ matrix of fresh variables is created. In the instance example:

```
Table = [ [v1, v2, v3, v4, v5], [v6, v7, v8, v9, v10], [v11, v12,
  v13, v14, v15], [v16, v17, v18, v19, v20], [v21, v22, v23, v24,
  v25], [v26, v27, v28, v29, v30], [v31, v32, v33, v34, v35] ]
```

p_tt_ta_2. abs is fit to $oabs$ (a list of lists representing the absences ordered per day). Also, atd (a list of lists representing the absences per day and team) and etd (a list of lists representing the request of ew per day and team) are created. In the instance example:

```
oabs = [ [1, 2, 5, 6, 7, 10, 11, 12], [], [], [], [],
                                      [5, 6, 7, 10, 11, 12], [] ];
atd = [ [2, 3, 3], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
                                          [0, 3, 3], [0, 0, 0] ];
etd = [ [1, 1, 1], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0],
                                          [0, 1, 1], [0, 0, 0] ];
```

p_tt_ta_3. The arrays of nd $\mathcal{FD}$ variables d ($d_i \equiv$ team assigned to dc_i), a ($a_i \equiv$ amount of regular worker absences on dc_i) and e ($e_i \equiv$ is ew requested on dc_i) are created. In the instance example:

```
d = [d1, d2, d3, d4, d5, d6, d7]
a = [a1, a2, a3, a4, a5, a6, a7]
e = [e1, e2, e3, e4, e5, e6, e7]
```

p_tt_ta_4. d is initialized to $\{1, \ldots, nt\}$. It is constrained by $d_i == d_{i+nt}$, and by an `all_different` constraint over $\{d_1, \ldots, d_{nt}\}$ to ensure that only one team works each day, and that each team is selected once every nt days. In the instance example:

```
d1 == d4; d4 == d7; d2 == d5; d3 == d6
{d1, d2, d3, d4, d5, d6, d7} in 1..3
{d1, d2, d3} are different
```

p_tt_ta_5. a is initialized to $\{0, \ldots, ntw\}$. The amount of shifts to be covered on each dc_i is used to prune the upper bound of each a_i. This precludes the assignment of a team to a day in case it does not provide enough available workers to accomplish the requested set of shifts. Also d, a and atd are linked, to constrain each a_i to be assigned to $atd_{i,di}$. In the instance example:

```
{a1, a2, a3, a4, a5, a6, a7} in 0..4
a1 <= 2; a2 <= 2; a3 <= 2; a4 <= 2; a5 <= 2; a6 <= 3; a7 <= 3;
(d1==1) => (a1==2); (d1==2) => (a1==3); (d1==3) => (a1==3);
(d2==1) => (a2==0); (d2==2) => (a2==0); (d2==3) => (a2==0);
(d3==1) => (a3==0); (d3==2) => (a3==0); (d3==3) => (a3==0);
(d4==1) => (a4==0); (d4==2) => (a4==0); (d4==3) => (a4==0);
(d5==1) => (a5==0); (d5==2) => (a5==0); (d5==3) => (a5==0);
(d6==1) => (a6==0); (d6==2) => (a6==3); (d6==3) => (a6==3);
(d7==1) => (a7==0); (d7==2) => (a7==0); (d7==3) => (a7==0);
```

p_tt_ta_6. e is initialized to $0, 1$. The sum of each array subset $\{e_i, \ldots, e_{i+er}\}$ is constrained to be ≤ 1, to ensure the resting constraint of ew. Also, d, e and etd are linked, to constrain that each e_i is assigned to $etd_{i,di}$. In the instance example:

```
{e1, e2, e3, e4, e5, e6, e7} in 0..1
sum [e1, e2, e3] <= 1, sum [e2, e3, e4] <= 1,
sum [e3, e4, e5] <= 1, sum [e4, e5, e6] <= 1,
sum [e5, e6, e7] <=1,
(d1==1) => (e1==1); (d1==2) => (e1==1); (d1==3) => (e1==1);
(d2==1) => (e2==0); (d2==2) => (e2==0); (d2==3) => (e2==0);
(d3==1) => (e3==0); (d3==2) => (e3==0); (d3==3) => (e3==0);
(d4==1) => (e4==0); (d4==2) => (e4==0); (d4==3) => (e4==0);
(d5==1) => (e5==0); (d5==2) => (e5==0); (d5==3) => (e5==0);
(d6==1) => (e6==0); (d6==2) => (e6==1); (d6==3) => (e6==1);
(d7==1) => (e7==0); (d7==2) => (e7==0); (d7==3) => (e7==0);
```

p_tt_ta_7. The variables $\{d_1, \ldots, d_{nt}\}$ are labeled to obtain the feasible $\{tda_1, \ldots, tda_k\}$. In the instance example:

```
label [d1, d2, d3] in order
```

5.3.2 Stage tt_split:

The call $tt\_split\ nd\ nt\ ntw\ ws\ oabs\ dc\ d\ e\ Table = (tt, dc\_list, h)$ runs the stage. In the instance example, just considering the first solution of $team\_assign\ tda_1$, the call results to be:

```
tt_split 7 3 4 [ [20, 22, 24], [24, 24] ] [ [1, 2, 5, 6, 7, 10,
  11, 12], [], [], [], [], [5, 6, 7, 10, 11, 12] ,[] ] [1, 2, 3,
  1, 2, 3, 1] [1, 0, 0, 0, 0, 1, 0] [ [v1, v2, v3, v4, v5], [v6,
  v7, v8, v9, v10], [v11, v12, v13, v14, v15], [v16, v17, v18,
  v19, v20], [v21, v22, v23, v24, v25], [v26, v27, v28, v29,
  v30], [v31, v32, v33, v34, v35] ] == (tt, dc_list, h)
```

A single solution is obtained:

```
tt = [ [ [0, 0, a, b, c], [l, m, n, o, 0], [v, w, x, y, 0] ],
       [ [d, e, f, g, 0], [p, q, r, s, 0] ],
       [ [h, i, j, k, 0], [t, 0, 0, 0, u] ] ]
dc_list = [ [1,1,2], [1,1], [1,2] ]
h = 35
```

The following steps are performed to compute the solution:

p_tt_sp_1. Receive $Table$, d, e and $oabs$ from $team\_assign$. In the instance example:

```
Table = [[v1,v2,v3,v4,v5],[v6,v7,v8,v9,v10],[v11,v12,v13,v14,
  v15],[v16,v17,v18,v19,v20],[v21,v22,v23,v24,v25],[v26,v27,
  v28,v29,v30],[v31, v32,v33,v34,v35]];
d = [1,2,3,1,2,3,1]; e = [1,0,0,0,0,1,0];
oabs = [[1,2,5,6,7,10,11,12],[],[],[],[],[5,6,7,10,11,12],[]]
```

p_tt_sp_2. $TotZ$ (a list of lists representing the selected worker absences per day) is computed, using d_i and $oabs_i$ to know which selected regular workers are absent on dc_i, and e_i to know if ew is selected for dc_i. In the instance example:

```
TotZ = [ [1,2], [5], [5], [5], [5], [2,3,4], [5] ]
```

p_tt_sp_3. For each dc_i $Table_i$ is traversed, binding to zero each variable indexed by $TotZ_i$. In the instance example:

```
Table = [[0,0,a,b,c],[d,e,f,g,0],[h,i,j,k,0],[l,m,n,o,0],
         [p,q,r,s,0],[t,0,0,0,u],[v,w,x,y,0]]
```

p_tt_sp_4. $Table$ is split into the independent subproblems $\{tt_1, \ldots, tt_{nt}\}$, each of them consisting of a matrix $\{(ntw + 1) \times (nd/nt)\}$. In the instance example:

```
tt = [ [ [0,0,a,b,c], [l,m,n,o,0], [v,w,x,y,0] ],
       [ [d,e,f,g,0], [p,q,r,s,0] ],
```

```
       [ [h,i,j,k,0], [t,0,0,0,u] ] ]
tt1 = [ [0,0,a,b,c], [l,m,n,o,0], [v,w,x,y,0] ]
tt2 = [ [d,e,f,g,0], [p,q,r,s,0] ]
tt3 = [ [h,i,j,k,0], [t,0,0,0,u] ]
```

dc is split as well, to obtain the day classification associated to each tt_i:

```
dc_list = [ [1,1,2], [1,1], [1,2] ]
dc_list1 = [1, 1, 2]
dc_list2 = [1, 1]
dc_list3 = [1, 2]
```

Finally, the number of hours that each regular worker is expected to work is computed. In the instance example, according to ws, each day of $dc \equiv ws_1$ (respectively ws_2) implies 66 (respectively 48) working hours. So, for $dc = [1, 1, 1, 1, 1, 2, 2]$ the timetable contains a total of 426 working hours. As there are 12 regular workers, each one is expected to work 35 hours:

```
h = 35
```

5.3.3 Stage tt_solve:

The call $tt\_solve\ ntw\ ws\ ef\ T\ h\ P\ W\ SS\ tt\ dc\_list = eh$ runs the stage. In the instance example, considering just the solving of tt_1 (from now on named tt), the call results to be:

```
tt_solve 4 [ [20, 22, 24], [24, 24] ] 2 1 35 true fst_unb
   workers [ [0, 0, a, b, c], [l, m, n, o, 0], [v, w, x, y, 0] ]
   [1, 1, 2] == eh
```

A single solution is obtained:

```
tt = [ [0, 0, 22, 24, 20], [0, 20, 24, 22, 0], [24, 24, 0, 0, 0] ]
eh = 71
```

The following steps are performed to compute the solution:

p_tt_so_1. tt is transposed to obtain $trans\_tt$, a matrix $\{(nd/nt) \times (ntw+1)\}$ ordered by workers instead of by days. In the instance example:

```
trans_tt = [ [0,l,v], [0,m,w], [a,n,x], [b,o,y], [c,0,0] ]
```

p_tt_so_2. For each day $dc_i = ws_j$ of tt the ws_j is parsed, obtaining the different shifts or working slots. These set of shifts is represented by their values (say v) and their cardinalities (say c). tt_i is initialized with domain v, and a global constraint is posted to ensure the distribution of the tt_i variables with (v, c). In the instance example:

```
{0, 0, a, b, c} in [0, 20, 22, 24]
distribute [2, 1, 1, 1] [0, 20, 22, 24] {0, 0, a, b, c}
{l, m, n, o, 0} in [0, 20, 22, 24]
distribute [2, 1, 1, 1] [0, 20, 22, 24] {l, m, n, o, 0}
{v, w, x, y, 0} in [0, 24]
distribute [3, 2] [0, 24] {v, w, x, y, 0}
```

p_tt_so_3. ls (a list of the different kind of shifts s_x to be scheduled in tt) is computed. In the instance example:

```
ls = [0, 20, 22, 24]
```

p_tt_so_4. ln (a mate list for ls, where each ln_i contains the number of shifts of type ls_i to be scheduled in tt) is computed. In the instance example:

```
ln = [7, 2, 2, 4]
```

p_tt_so_5. For each ls_i new ($ntw + 1$) $\mathcal{FD}$ variables cv are created, where each cv_k is assigned to the amount of shifts of type ls_i the worker $trans\_tt_k$ is assigned to. The sum of these cv variables is redundantly constrained to ln_i, to ensure that all the shifts are assigned to a worker. In the instance example:

```
count 0 [0, l, v] == cv0_1; count 0 [0, m, w] == cv0_2;
count 0 [a, n, x] == cv0_3; count 0 [b, o, y] == cv0_4;
count 0 [c, 0, 0] == cv0_5;
sum [cv0_1, cv0_2, cv0_3, cv0_4, cv0_5] == 7

count 20 [0, l, v] == cv20_1; count 20 [0, m, w] == cv20_2;
count 20 [a, n, x] == cv20_3; count 20 [b, o, y] == cv20_4;
count 20 [c, 0, 0] == cv20_5;
sum [cv20_1, cv20_2, cv20_3, cv20_4, cv20_5] == 2

count 22 [0, l, v] == cv22_1; count 22 [0, m, w] == cv22_2;
count 22 [a, n, x] == cv22_3; count 22 [b, o, y] == cv22_4;
count 22 [c, 0, 0] == cv22_5;
sum [cv22_1, cv22_2, cv22_3, cv22_4, cv22_5] == 2

count 24 [0, l, v] == cv24_1; count 24 [0, m, w] == cv24_2;
count 24 [a, n, x] == cv24_3; count 24 [b, o, y] == cv24_4;
count 24 [c, 0, 0] == cv24_5;
sum [cv24_1, cv24_2, cv24_3, cv24_4, cv24_5] == 4
```

p_tt_so_6. The fairness of the distribution is tightened with T by constraining the differences of cv to be in the domain $\{-T, \ldots, T\}$. In the instance example:

```
aux0_1 == cv0_1 - cv0_2; aux0_2 == cv0_1 - cv0_3;
aux0_3 == cv0_1 - cv0_4; aux0_4 == cv0_2 - cv0_3;
aux0_5 == cv0_2 - cv0_4; aux0_6 == cv0_3 - cv0_4;
{aux0_1, aux0_2, aux0_3, aux0_4, aux0_5, aux0_6} in (-1)..1

aux20_1 == cv20_1 - cv20_2; aux20_2 == cv20_1 - cv20_3;
aux20_3 == cv20_1 - cv20_4; aux20_4 == cv20_2 - cv20_3;
aux20_5 == cv20_2 - cv20_4; aux20_6 == cv20_3 - cv20_4;
{aux20_1, aux20_2, aux20_3, aux20_4, aux20_5, aux20_6} (-1)..1

aux22_1 == cv22_1 - cv22_2; aux22_2 == cv22_1 - cv22_3;
aux22_3 == cv22_1 - cv22_4; aux22_4 == cv22_2 - cv22_3;
aux22_5 == cv22_2 - cv22_4; aux22_6 == cv22_3 - cv22_4;
{aux22_1, aux22_2, aux22_3, aux22_4, aux22_5, aux22_6} (-1)..1

aux24_1 == cv24_1 - cv24_2; aux24_2 == cv24_1 - cv24_3;
aux24_3 == cv24_1 - cv24_4; aux24_4 == cv24_2 - cv24_3;
aux24_5 == cv24_2 - cv24_4; aux24_6 == cv24_3 - cv24_4;
{aux24_1, aux24_2, aux24_3, aux24_4, aux24_5, aux24_6} (-1)..1
```

p_tt_so_7. The extra hours of tt is computed via the sum of the hours of the regular workers (implication constraints are used to take into account only those workers performing more than h hours) plus the sum of the hours of $ew * ef$. In the instance example:

```
sum [0, l, v] == hoW1; hoW1 - 35 == eWL1;
(eWL1 >= 0) => (eH1 == eWL1); (eWL1 < 0) => (eH1 == 0);

sum [0, m, w] == hoW2; hoW2 - 35 == eWL2;
(eWL2 >= 0) => (eH2 == eWL2); (eWL2 < 0) => (eH2 == 0);

sum [a, n, x] == hoW3; hoW3 - 35 == eWL3;
(eWL3 >= 0) => (eH3 == eWL3); (eWL3 < 0) => (eH3 == 0);

sum [b, o, y] == hoW4; hoW4 - 35 == eWL4;
(eWL4 >= 0) => (eH4 == eWL4); (eWL4 < 0) => (eH4 == 0);

sum [eWL1, eWL2, eWL3, eWL4] == tot_EWL;
sum [c, 0, 0] == eWL5
eh = tot_EWL + (2 * eWL5)
```

p_tt_so_8. tt variables are labeled to find the assignment that minimizes the extra hours. In the instance example:

```
label [0,l,v,0,m,w,a,n,x,b,o,y,c,0,0] in order, minimizing eh
```

5.3.4 Stage tt_map:

The call $tt\_map\ nt\ ntw\ d\ tt$ = $timetabling$ runs the stage. In the instance example, the call results to be:

```
tt_map 3 4 [1, 2, 3, 1, 2, 3, 1] [ [0, 0, 22, 24, 20], [0, 20,
  22, 24, 0], [0, 20, 22, 24, 0], [0, 20, 24, 22, 0], [24, 22,
  20, 0, 0], [24, 0, 0, 0, 24], [24, 24, 0, 0, 0] ]
                                                    == timetabling
```

A single solution is obtained:

```
timetabling = [ [ 0, 0, 22, 24, 0, 0, 0, 0, 0, 0, 0, 0, 20 ],
                [ 0, 0, 0, 0, 0, 20, 22, 24, 0, 0, 0, 0, 0 ],
                [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 20, 22, 24, 0 ],
                [ 0, 20, 24, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0 ],
                [ 0, 0, 0, 0, 24, 22, 20, 0, 0, 0, 0, 0, 0 ],
                [ 0, 0, 0, 0, 0, 0, 0, 0, 24, 0, 0, 0, 24 ],
                [ 24, 24, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ]
```

The following steps are performed to compute the solution:

p_tt_ma_1. The solved $Table$ is received, mapping it to $(timetabling, eh)$.

5.4 Performance

The modeling and solving of the ETP in $\mathcal{TOY}(\mathcal{FD})$ and other state-of-the-art CP($\mathcal{FD}$) systems is deeply analyzed in Chapters 7 and 8, respectively. However, from the solving point of view, the ETP results are compared first with the ones of Sections 3.4 and 4.3, discussing if the conclusions for solving classical CP($\mathcal{FD}$) benchmarks also hold for solving the real-life ETP. This section focuses on this comparison. The two $\mathcal{TOY}(\mathcal{FD})$ models $p\_tt\_bs.toy$ and $p\_tt\_is.toy$ (respectively used for Sections 5.4.1 and 5.4.2) are available at: `http://gpd.sip.ucm.es/ncasti/models.zip`.

5.4.1 Performance of the Different $\mathcal{TOY}(\mathcal{FD})$ Versions

As it happened with the benchmarks of Section 3.4, it is claimed now that the ETP just formulated is representative enough for testing the performance of $\mathcal{TOY}(\mathcal{FD})$.

First, it is also a parametric problem, and thus the three different instances ETP-7, ETP-15 and ETP-21 (whose solving times for all the $\mathcal{TOY}(\mathcal{FD})$ versions are of tenths of seconds, seconds and minutes, respectively) have been selected. These three instances follow the same input parameters as the instance example presented in Section 5.1,

varying just the parameter nd in 7, 15 and 21, to consider a timetable of one, two and three weeks, respectively. The achieved solving times are relevant for comparing the results with those of Table 3.1, turning ETP-7 into a mate instance for M-400, Q-90, L-119 and G-9; ETP-15 into a mate instance for M-900, Q-105, L-127 and G-10; ETP-21 into a mate instance for Q-120, L-131 and G-11.

Second, the ETP also includes the whole set of $\mathcal{FD}$ constraints of the $\mathcal{TOY}(\mathcal{FD})$ repertoire. More specifically: Relational constraints, as #<=, posted in p_tt_ta_5 (cf. Section 5.3). Arithmetic constraints, as #-, posted in p_tt_so_6. Propositional constraints, as `post_implication`, posted in p_tt_ta_5 and p_tt_so_7. Domain constraints, as `domain` and `domain_valArray`, posted in p_tt_ta_3 and p_tt_so_2, respectively. Global constraints, as `all_different`, `distribute`, `count` and `sum`, posted in p_tt_ta_4, p_tt_so_2, p_tt_so_5 and p_tt_so_6, respectively. Finally, both stages $team\_assign$ and tt_solve end with a `labeling` search strategy. Whereas the former relies on a satisfiability search exploration (to find any feasible tda) the latter relies on an optimization one (to find the tt assignment minimizing the amount of extra hours).

Table 5.2 presents the results for running the ETP instances in the different $\mathcal{TOY}(\mathcal{FD})$ versions. These results are compared to the ones of Table 3.1. Column `Instance` represents the instance being run (the concrete $\mathcal{TOY}(\mathcal{FD})$ version solving it is represented by FDs, FDg and FDi for $\mathcal{TOY}(\mathcal{FD}s)$, $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$, respectively). Next block of three columns represent results for incremental propagation mode. Column `Incremental` represents the CPU solving time, measured in milliseconds. Column `Perc_I` represents the percentage of the CPU solving time devoted to $\mathcal{FD}$ search exploration. Column Sp-Up_I represents the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$. Next block of three columns are the same, but for batch propagation. Finally, column `I/B` represents the speed-up of using batch mode instead of incremental one. The following conclusions are drawn:

- The approach of enhancing the $\mathcal{TOY}(\mathcal{FD})$ performance by focusing on its $\mathcal{FD}$ solver is still encouraging for $\mathcal{TOY}(\mathcal{FD}g)$, but this is not as clear when considering $\mathcal{TOY}(\mathcal{FD}i)$. Thus, both versions are discussed separately.

Instance	Incremental	Perc_I	Sp-Up_I	Batch	Perc_B	Sp-Up_B	I/B
ETP-7 FDs	0.248	35.5	1.00	0.248	37.6	1.00	1.00
ETP-7 FDg	0.192	2.1	1.30	0.186	4.0	1.33	1.03
ETP-7 FDi	1.380	0.3	0.18	0.998	0.5	0.25	1.39
ETP-15 FDs	3.22	91.9	1.00	2.72	92.7	1.00	1.19
ETP-15 FDg	0.90	60.0	3.57	0.88	63.3	3.13	1.02
ETP-15 FDi	3.28	51.2	0.98	2.56	69.9	1.06	1.28
ETP-21 FDs	338.16	99.9	1.00	261.68	99.9	1.00	1.25
ETP-21 FDg	50.04	99.1	6.67	49.92	99.1	5.26	1.00
ETP-21 FDi	109.02	98.6	3.13	107.96	99.3	2.44	1.01

Table 5.2: ETP Performance Results

Regarding $\mathcal{TOY}(\mathcal{FD}g)$, it improves the performance w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$ for all the ETP instances, in a range of 1.30-6.67 times faster. This range is even wider than the 1.37-3.57 achieved for the classical CP($\mathcal{FD}$) benchmarks (where four different problems were run).

Moreover, the improvement of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$ increases as the instances scale up. This already happened for Queens and Langford's (a 1.81-2.50 improvement for Q-90 to a 3.33-3.57 one for Q-105 and Q-120; a 1.37 improvement for L-119 to a 2.56-2.63 one for L-127 and L-131) but, once again, in ETP this improvement increase is wider (with a 1.30-1.33 for ETP-7, a 3.13-3.57 for ETP-15 and a 5.26-6.67 for ETP-21). As it can be seen, this also includes a gap between the instance solved in seconds and the instance solved in minutes (which did not happen neither in Queens nor in Langford's).

Regarding $\mathcal{TOY}(\mathcal{FD}i)$, three different situations happen for solving the three ETP instances. First, for ETP-7, its performance (w.r.t. the one of $\mathcal{TOY}(\mathcal{FD}s)$) is clearly worse, ranging in 4.0 and 4.5 times slower. Second, for ETP-15, its performance is similar to the one of $\mathcal{TOY}(\mathcal{FD}s)$ (ranging in a 0.98-1.06 times better). Finally, for ETP-21, its performance clearly improves (a 2.44-3.13 times faster) the one of $\mathcal{TOY}(\mathcal{FD}s)$. Thus, as it happened with $\mathcal{TOY}(\mathcal{FD}g)$, it is clear than the improvement of $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$ increases as the instances scale up, even much more than it did for Queens and Langford's instances (where the improvement increased from a 1.30-1.81 for Q-90 to a 2.78-2.94 for Q-105 and Q-120; 1.37 for L-119 to a 2.78-2.94 for L-127 and L-131).

- The clear correlation between the time the $\mathcal{TOY}(\mathcal{FD})$ versions devote to search exploration and the improvement $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ achieve w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$ still remains for ETP. Moreover, it justifies the different behavior (w.r.t. classical CP($\mathcal{FD}$) benchmarks) presented before.

  ETP behaves as Queens and Langford's, where the CPU time devoted to search exploration increases as the instances scale up. However, whereas $\mathcal{TOY}(\mathcal{FD}s)$ maintains a similar increasing pattern for ETP, Queens and Langford's, in $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ the increasing pattern for ETP is quite different.

  This leads to two mismatches between $\mathcal{TOY}(\mathcal{FD}s)$ and both $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$: First, the percentage devoted to search exploration of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ for ETP-7 (2.1%-4% and 0.3%-0.5%, respectively) is much smaller than the one for $\mathcal{TOY}(\mathcal{FD}s)$ (35.5%-37.6%). Moreover, this small percentage of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ is not even comparable to the ones they achieve either for Q-90 and L-119. A second mismatch is that the percentage devoted to search exploration of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ for ETP-15 (60%-63% and 51%-69%, respectively) is much smaller than the one for $\mathcal{TOY}(\mathcal{FD}s)$ (92%-93%). Moreover, this small percentage of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ is not even comparable to the ones they achieve either for Q-105 and L-127, but comparable to

the ones for Q-90 and L-119 (whose solving times are an order of magnitude smaller than the ones for ETP-15).

With these mismatches for ETP-7 and ETP-15, it is normal than the performance improvement of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ (w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$) differs from the ones for Q-90 and Q-105 (respectively L-119 and L-127), as the search exploration times are not comparable. Also, as in $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ the percentage of time devoted to search exploration clearly increases from ETP-15 to ETP-21, the improvement achieved also increases (in contrast to Q-105 and Q-120 or L-127 and L-131, where both the percentage of search exploration and the improvement achieved remain more or less stable).

- The selected propagation mode becomes relevant for ETP, as the differences achieved between batch mode (faster for all the instances) and incremental one are much greater than those achieved when solving classical CP($\mathcal{FD}$) benchmarks. This does not happen for $\mathcal{TOY}(\mathcal{FD}g)$, where the 6ms, 20ms and 280ms differences respectively achieved for ETP-7, ETP-15 and ETP-21 (cf. columns `Incremental` and `Batch`) are always two orders of magnitude smaller than the CPU solving time. However, in $\mathcal{TOY}(\mathcal{FD}i)$, the 382ms and 720ms differences for ETP-7 and ETP-15 represent a 1.39 and a 1.28 (respectively) improvement of batch w.r.t incremental. And, finally, in $\mathcal{TOY}(\mathcal{FD}s)$, the 500ms and 76.480ms differences for ETP-15 and ETP-21 represent a 1.19 and 1.25 (respectively) improvement of batch w.r.t. incremental. The latter is specially remarkable, as it represents the difference between solving the problem in 4.5 minutes or in 5.5 minutes.

In summary, it can be seen that the performance results of ETP are similar to the ones of Queens and Langford's. However, ETP is said to be a more extreme problem, where the performance results are much more dependent on the concrete instance being run than when running any classical CP($\mathcal{FD}$) benchmark.

5.4.2 Performance of Applying the Search Strategies

The bs model uses in p_tt_so_8 a single `labeling [toMinimize eh] TransTT` as its search strategy. Analyzing the solutions of the three instances ETP-7, ETP-15 and ETP-21 being proposed (more specifically, analyzing the solutions of tt_1, tt_2 and tt_3 for the two feasible tda_1 and tda_2, as well as the initial search space of these tt_1, tt_2 and tt_3) it is observed that there is a higher pruning of the search space if some of the weekend variables are labeled first to the highest value (24) of their domains. Intuitively, it makes sense, as weekend variables have a smaller domain {0, 24} than working days ones {0, 20, 22, 24}. Thus, due to the `distribute` constraint posted on the variables of each day (see p_tt_so_2), the binding of some variables in a weekend day may lead to the binding of the whole working team for that concrete day. However, if this is

not the case, due to the redundant `count` constraint posted on each kind of shift (see p_tt_so_5) and to the structure of the solutions (which contain more 0 hour shifts than 24 ones), the binding of some weekend variables to 24 prunes more the search tree than the binding of those same variables to 0.

Thus, the new search strategy of the is model applies `labB smallestDomainVar largestVal 2 (foldl (++) [] TransTT)` first. This binds to 24 two of the weekend variables of each team tt_i, highly reducing the search space the further optimization `labeling` has to deal with (although it might cause a loss of completeness). Also, as `labB` precludes any further backtracking to explore other bindings for those weekend variables, the optimization search exploration is triggered just once per each team tt_i (as it happened in bs when using the original search strategy).

Table 5.3 presents the results for running the ETP instances in $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ by using the is models. These results are compared to the ones of Table 4.2. Columns `bs` and `is` represent the CPU solving time of bs and is (respectively), both of them using incremental propagation mode. Columns `Sp-Up_bs` and `Sp-Up_is` represent the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ for bs and is, respectively. Finally, column `off/on` focuses on each concrete $\mathcal{TOY}(\mathcal{FD})$ version, representing the speed-up of is w.r.t. bs. The following conclusions are drawn:

- The use of the new search strategies is encouraging, as the performance of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ for solving is instances is better than the achieved for solving bs ones. In this setting, once again, the behavior of ETP is similar to the one of Queens and Langford's, as the better performance achieved clearly scales as the size of the instances scale. However, the impact of applying the new search strategies is smaller in ETP than in Queens and Langford's. This makes sense, at least for ETP-7 and ETP-15, as their devoted time to search exploration are much smaller than the ones for Q-90, Q-105, L-119 and L-127.

  In ETP-7 both $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ have a worse performance in is than in bs. However, whereas in $\mathcal{TOY}(\mathcal{FD}i)$ the 0.97 worst performance is smaller than the 0.21 and 0.54 worst performance $\mathcal{TOY}(\mathcal{FD}i)$ achieved for Q-90 and L-119 (respectively), in $\mathcal{TOY}(\mathcal{FD}g)$ the 0.95 worst performance supposes a difference w.r.t. the 1.28 and 1.05 improvement $\mathcal{TOY}(\mathcal{FD}g)$ achieved for Q-90 and L-119, respectively. In any case, in Table 5.2, it has been seen that the percentage of CPU time devoted to search exploration of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ for ETP-7 is just 2.1% and 0.3%, respectively. Thus, the reason of this 0.97 and 0.95 worse performance for is relies on the variable selection method `smallestDomainVar` (i.e., first fail) of `labB`, whose implementation in the search strategy infrastructure of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ is much less optimized than the one in the original Gecode and ILOG Solver API.

  In ETP-15 and ETP-21, this overhead for the variable selection method still plays its role, but the high pruning of the search space achieved by `labB` overcomes

Instance	bs	Sp-Up_bs	is	Sp-Up_is	off/on
ETP-7 FDi	1.380	1.00	1.420	1.00	0.97
ETP-7 FDg	0.192	7.14	0.202	7.14	0.95
ETP-15 FDi	3.28	1.00	2.64	1.00	1.25
ETP-15 FDg	0.90	3.70	0.48	5.56	1.89
ETP-21 FDi	109.02	1.00	22.90	1.00	4.76
ETP-21 FDg	50.04	2.17	9.56	2.38	5.26

Table 5.3: ETP Performance Using the Search Strategies

it. Whereas $\mathcal{TOY}(\mathcal{FD}g)$ achieves a 1.89 and 5.26 improvement for ETP-15 and ETP-21 (respectively), $\mathcal{TOY}(\mathcal{FD}i)$ achieves a 1.25 and 4.76.

- Second, the fact that the improvement achieved by $\mathcal{TOY}(\mathcal{FD}g)$ for is is greater than the one achieved by $\mathcal{TOY}(\mathcal{FD}i)$ still remains for ETP. However, whereas for Queens and Langford's the gap between $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ for is instances increased as the instances scale up, for ETP it does not (0.97 and 0.95 of $\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}g)$ for ETP-7, respectively; 1.25 and 1.89 for ETP-15; 4.76 and 5.26 for ETP-21).

5.5 Related Work

As it was said, the ETP being proposed can be seen as a particular case of the NRP, the most paradigmatic example of an Employee Timetabling Problem, in which an assignment of the nurse roster to the working shifts of a hospital must be scheduled. The NRP has been extensively studied in the last decades, and different Mathematical Programming, Heuristics and CP($\mathcal{FD}$) approaches have been tried to tackle it [38].

Within Mathematical Programming, recent applications include Linear Programming [200] (applied to a hospital in The Netherlands), Integer Programming [164] (applied to the International Nurse Rostering Competition [144]) and Mixed Integer Programming [162] (applied to a typical Swedish nursing ward). Within Heuristics recent applications include Evolutionary Algorithms [204] (applied to a typical hospital environment) and Hybrid Methods [21] (hybridizing an stochastic ranking method with a Simulated Annealing heuristic within a Local Search and a Genetic Algorithm framework, and applying it to a problem based on a hospital in United Kingdom).

Within CP($\mathcal{FD}$) recent applications include C++ CP($\mathcal{FD}$) [152] (using ILOG Solver to find satisfiability solutions to a problem based on a hospital in The Netherlands) and the use of *ad hoc* soft constraints [137], which are integrated into a Variable Neighborhood Search [89] with reconstruction based on Limited Discrepancy Search [95] (applied to

the instances of the University of Nottingham). Finally, regarding CFLP($\mathcal{FD}$) there are no previous applications of NRP, but so there are for other timetabling problems [32].

5.6 Conclusions

The classical CP($\mathcal{FD}$) benchmarks used in Chapters 3 and 4 have shown that interfacing external C++ CP($\mathcal{FD}$) solvers and applying *ad hoc* search strategies have improved the solving performance of $\mathcal{TOY}(\mathcal{FD})$. However, the modeling of these problems is quite simple, and thus they have not exploited the high expressivity of $\mathcal{TOY}(\mathcal{FD})$. Thus, in this chapter a real-life ETP coming from the communication industry has been proposed, with the aim of exploiting both the high expressivity of $\mathcal{TOY}(\mathcal{FD})$ and its higher solving performance just achieved.

A description of the ETP has been presented, which is parametric in the number of days of the timetable, number of teams (and number of workers per team), periodicity the extra worker can be selected (and the extra factor its working hours must be paid), number of different kinds of working days (and the concrete shifts requested on each of them), absences of the regular workers of the teams and the distribution of the shifts among the workers of a team. Also, an instance example has been proposed to clarify all these concepts. Then, a solving approach to tackle the ETP has been presented, which is based on splitting the initial search space into as many different assignments of teams to days there are, exploring only those ones that are feasible and, for each of them, decomposing this search space subset into as many independent problems (exponentially easier to be solved) as teams there are.

Then, a parametric timetabling algorithm implementing the solving approach has been described. It is based on a four stage process: $team\_assign$, tt_split, tt_solve and tt_map. First stage $team\_assign$ just concerns with finding any feasible assignment of teams to days. Starting from it, the stages tt_split, tt_solve and tt_map are executed, decomposing this search space subset into the independent subproblems, solving them and fitting the format of the solution, respectively. The sequence of stages is repeated for each feasible assignment. Finally, the computed solutions are compared, outputting as a result the one with minimum extra hours. The stages of the algorithm have been presented separately, enumerating for each of them the steps being followed and instantiating those steps to the instance example.

Finally, the results of Sections 3.4 and 4.3 have been revisited, and the ETP-7, ETP-15 and ETP-21 instances have been used (whose solving times are as well of tenths of seconds, seconds and minutes, respectively) to discuss if the conclusions for solving classical CP($\mathcal{FD}$) benchmarks also hold for solving the real-life ETP. It has been shown that ETP behaves similarly to Queens and Langford's: The $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ improvement w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$ increases as the instances scale up, as well as the CPU solving time devoted to search exploration and the impact of the improved search strat-

egy applied.

However, the ETP has been claimed to be a more extreme problem, where the results are more dependent on the concrete instance being run than when solving Queens or Langford's. There are some remarkable issues to be pointed out:

- $\mathcal{TOY}(\mathcal{FD}g)$ still outperforms $\mathcal{TOY}(\mathcal{FD}s)$, but the range of its improvement achieved is wider than the one for Queens and Langford's.
- $\mathcal{TOY}(\mathcal{FD}i)$ behaves respectively worse, equivalent or better than $\mathcal{TOY}(\mathcal{FD}s)$ for the three ETP instances.
- The search exploration percentage of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ for ETP-7 and ETP-15 is much smaller than the one for $\mathcal{TOY}(\mathcal{FD}s)$, and the ones of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ for the mate Queens and Langford's instances.
- The batch propagation mode clearly improves incremental mode in $\mathcal{TOY}(\mathcal{FD}i)$ for ETP-7 and ETP-15, and in $\mathcal{TOY}(\mathcal{FD}s)$ for ETP-15 and ETP-21 (the latter achieving a remarkable difference of minutes).
- The impact of applying the improved search strategy is smaller in ETP instances than in Queens and Langford's instances. Moreover, the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ does not increase as the instances scale up.

Chapter 6

Bin Packing Problem

The application of $\mathcal{TOY}(\mathcal{FD})$ to the real-life ETP of Chapter 5 has exploited all the expressive power of the system, as well as the better solving performance achieved (by using the techniques of Chapters 3 and 4). In this case, the relation of $\mathcal{TOY}(\mathcal{FD})$ with the real-life problem to be tackled has been direct: A department needed to schedule a timetabling, and $\mathcal{TOY}(\mathcal{FD})$ was applied to model and solve it.

In this chapter, a second real-life application of $\mathcal{TOY}(\mathcal{FD})$ is presented. But, in this case, the relation of the system with the real-life problem to be tackled is indirect: $\mathcal{TOY}(\mathcal{FD})$ is applied to the classical Bin Packing Problem [2] (from now on just BPP), contributing to the future development of portfolio solvers to tackle configuration problems (which can be seen as generalizations of the BPP) coming from the industry of the data centre optimization. A more detailed explanation of the entire process and the role $\mathcal{TOY}(\mathcal{FD})$ plays on it is given next.

The classical BPP can be defined as follows: Given a set $S = \{s_1, \ldots, s_n\}$ of n indivisible items (each of a known positive size s_i) and a number of bins (each of capacity C), the goal is to pack the items into the minimal number m of bins (such that the sum of sizes of the items in each bin does not exceed C).

A data centre have to tackle several configuration problems, which can be seen as BPP's as they consist of assigning a group of objects to a set of containers [155]. For example, workload consolidation involves ensuring that the total amount of resources required by the set of jobs assigned to a server does not exceed the capacity of that resource. An even simpler example is the assignment of the servers to power units. Unfortunately, these configuration problems are more general than the classical BPP, as minimizing the number of servers or power units can cost less energy, but other side effects as agility, reliability or sustainability (among others) must be considered. Moreover, the success on the application of a concrete Mathematical Programming, Heuristics or CP($\mathcal{FD}$) approach to the problem is dependent on the concrete instance to be solved (that is, the application of a same technique to the solving of two different instances can lead to very different performance results). In summary, computing

optimized deployment plans requires to solve a hard planning problem and that plan may be subject to the addition of side requirements that cannot be defined in advance. Therefore, a robust method is needed to tackle it.

In this chapter an empirical analysis of the solving hardness of the classical BPP is presented. It will serve as the basis for the future development of portfolio solvers, useful for tackling generalized BPP real-life problems arisen in the optimization of a data centre. In the empirical analysis, both heuristics and CP($\mathcal{FD}$) systems (the latter including $\mathcal{TOY}(\mathcal{FD}g)$) are applied to the solving of a parametric generated benchmark of BPP instances, which is of particular interest as it accurately fits different aspects of the real-life generalized BPP instances. The experiments rely on first fixing the set of algorithms proposed (heuristics and CP($\mathcal{FD}$) models), and then running the instances of the benchmark, analyzing the existing relations between the parameters of the generated instances and the success of the different techniques for solving them. The conclusions provide a basis for the future development of portfolio-based solvers for the generalized BPP instances. These solvers will rely on the structure of an instance (the parameters that generated it) to learn their best configuration and apply the most suitable technique for solving it.

The chapter is organized as follows: Section 6.1 discusses the parametrical statistical model provided by the Weibull distribution [201], showing the variety of item size distributions that can be generated with it. It proves the model to be successful on fitting real-life generalized BPP instances coming from the data centre optimization and education industries. Section 6.2 presents the empirical analysis process layout. It describes the set of instances created (which gives support for very controlled experiments) and both the heuristics and CP($\mathcal{FD}$) models used to run the instance set. Section 6.3 analyzes the results by the CP($\mathcal{FD}$) and heuristics methods, focusing on the relation between the Weibull parameters of the BPP instances and the quality of the solutions. Section 6.4 presents some related work. Finally, Section 6.5 reports conclusions.

6.1 Weibull-Based Benchmark Approach

The relevance of the BPP (it is a ubiquitous problem that arises in many practical applications) and its difficulty to be solved (it is NP-complete) turns it into a deeply studied problem. The literature contains numerous approaches to tackle it and, interestingly, each proposed method attaches its own benchmark to show its performance [70, 115, 168, 175, 176, 199]. Thus, a first difficulty to carry out the empirical analysis proposed in this chapter comes from the lack of standardized benchmarks for BPP.

Moreover, it is a must for the benchmark finally proposed to be representative enough of the real-life generalized BPP instances, as the conclusions in the empirical analysis will be applied to the further development of specialized techniques for tackling these real-life instances. A big gap between the benchmark and the real-life

instances would make the empirical analysis meaningless. Unfortunately, the different BPP benchmark suites proposed in the literature are all artificial and lacking a practical basis. Typically, as for example in the benchmarks by Scholl and Klein [167], item sizes are generated using either uniform or normal distributions. As it was pointed out in [80], current benchmark suites in this area are often unrealistic and trivial to solve. Thus, the use of more realistic BPP benchmark suites for their study in large-scale data centre problems are demanded [155].

In this chapter a new BPP benchmark suite is proposed, which is based on the well known Weibull parametric continuous probability distribution. It is named after Waloddi Weibull, who presented the distribution in a seminal paper in 1951 [201], and is defined by both a shape ($k > 0$) and a scale ($\lambda > 0$) parameters. Figure 6.1 presents the probability density function, $f(x; \lambda, k)$, of a random variable x distributed according to Weibull. The parameters give rise to a great flexibility, allowing to represent many distributions that naturally occur in a variety of problem domains involving distributions of time horizons, time slots or lot sizes [201]. Figures 6.2 and 6.3 present several examples of different distributions that can be obtained by instantiating the Weibull distribution. The former presents four different distributions for small values (0.5, 1.0, 1.5 and 5.0) of the shape parameter, k. Clearly very different regimes are possible, some exhibiting extremely high skew around the value specified by the distribution's scale parameter, λ. In the latter, larger values of the shape parameter (9.0, 12.0, 15.0 and 18.0) are considered, and it can be seen that the distribution exhibits lower variation as shape increases.

A BPP benchmark suite based on the Weibull distribution successfully represents the real-life generalised BPP instances coming from the industry of the data centre optimization. To ensure it, a first subsection shows, visually, the quality of the fit that can be obtained. Then, a second subsection presents a more rigorous analysis of the goodness of fit by using two standard statistical tests.

6.1.1 Fitting Data Centre Real-life Instances

The 2012 ROADEF/EURO Challenge [1] is concerned with the problem of machine reassignment, with data and sponsorship coming from Google. The subject of the challenge is to find a best-cost mapping of processes, which have specific resource requirements,

$$f(x; \lambda, k) = \begin{cases} \frac{k}{\lambda} \cdot (\frac{x}{\lambda})^{k-1} \cdot e^{-(x/\lambda)^k} & x \geq 0, \\ 0, & \text{otherwise} \end{cases}$$

Figure 6.1: Weibull Probability Density Function

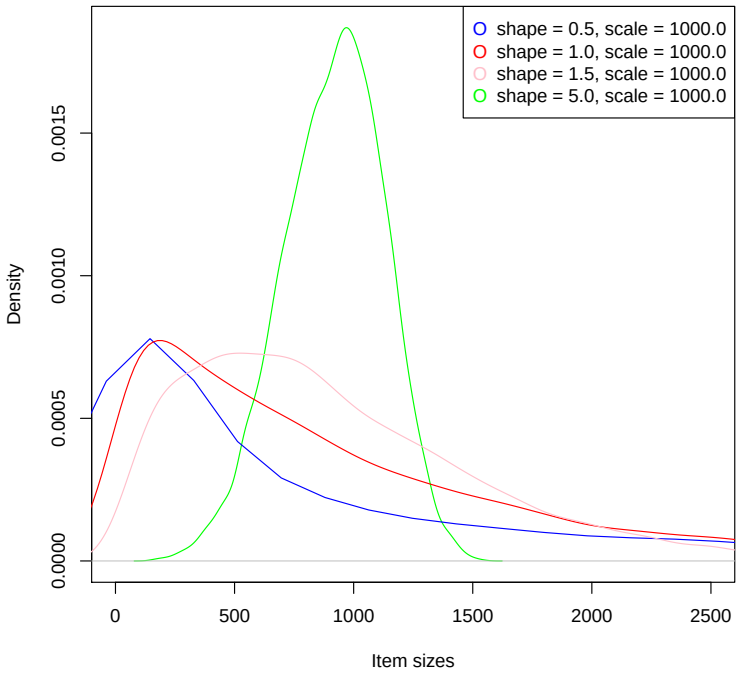

Figure 6.2: Weibull Distributions with Small Shape Parameters

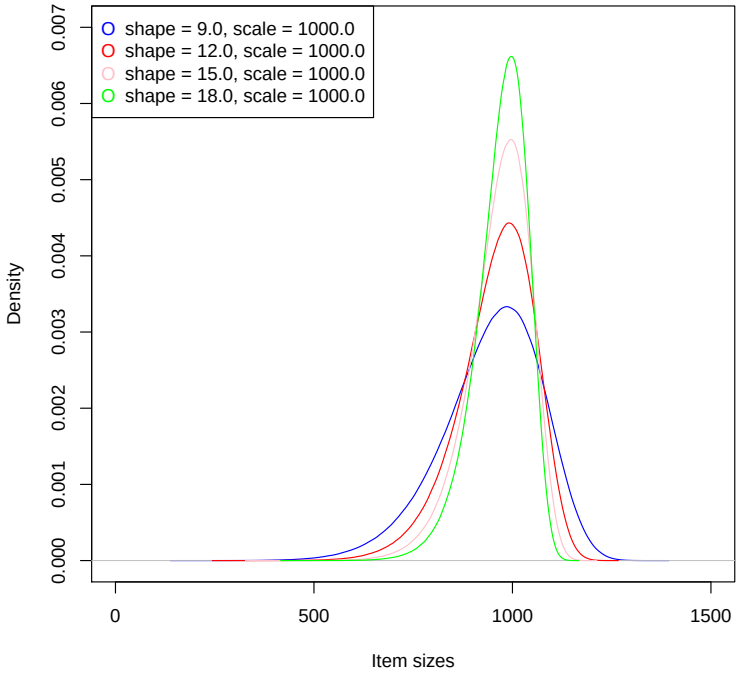

Figure 6.3: Weibull Distributions with Large Shape Parameters

onto machines, such that a variety of constraints is satisfied. A core element of the problem are bin packing constraints stating that the total amount of a given resource required by the processes assigned to a machine does not exceed the amount available. An important element of this challenge is the mapping of processes to machines such that the availability of each resource on the machine is not exceeded by the requirements of the set of services assigned to it. This subproblem is a multi-capacity BPP: Each machine is a bin with many elements defined by the set of resources available, and each process corresponds to an item that consumes different amounts of each resource. A publicly available set of problem instances that contains many BPP is provided. In particular, the instance a2(5) is used to show how well a Weibull-based benchmark model fits the BPP of this challenge.

Figure 6.4 presents the probability distribution for the resource 10 of this a2(5) instance. The probability density function that corresponds to the actual data is plotted as a line. It is clearly seen that the distribution is extremely skewed, with the majority of the probability mass coming from smaller items (values from 0 and 20,000). Then, the range of likely item sizes spans several orders of magnitude (over 100,000), and there is a very small possibility of encountering extremely large items.

R, the open-source statistical computing platform [8], has been used to fit a Weibull distribution to the data of this resource, by using Maximum Likelihood Fitting (MLF) [186]. Specifically, the R Weibull Distribution MLF implementation by Wessa has been used, which is available as an online service [202]. The resulting Weibull is presented in Figure 6.4 as the circles imposed on the density function from the data. A deeper visual analysis of the quality of the fit can be done by using a Quantile-Quantile plot (Q-Q plot). These plots are a simple tool to determine whether two data sets come from the same underlying distribution. In a Q-Q plot, each point corresponds to a particular quantile from both data sets. If, in the resultant plot, the set of points sit on a line of slope 1 then it can be concluded that the underlying distributions are the same. Figure 6.5 presents the Q-Q plot (generated by R) for the resource 10 of a2(5).

By observing both figures, it can be seen that the Weibull distribution computed by MLF fits very accurately the actual data of the instance. In particular, for the values ranging from 0 to 20,000 (which, once again, represent the majority of the probability mass) the fit is quite close. Then, for the outliers values spanning several orders of magnitude the fitting is worse, but it makes sense, as the probability of this values is nearly 0. However, besides this graphical observation, to ensure more rigorously the quality of the fitting of Weibull to a2(5) and other real-life instances, next section uses two statistical tests.

6.1.2 Verifying the Goodness of Fit

The whole set of instances from the 2012 ROADEF/EURO Challenge are gathered. In addition to those, a set of real-life examination timetabling (from now on just ETT)

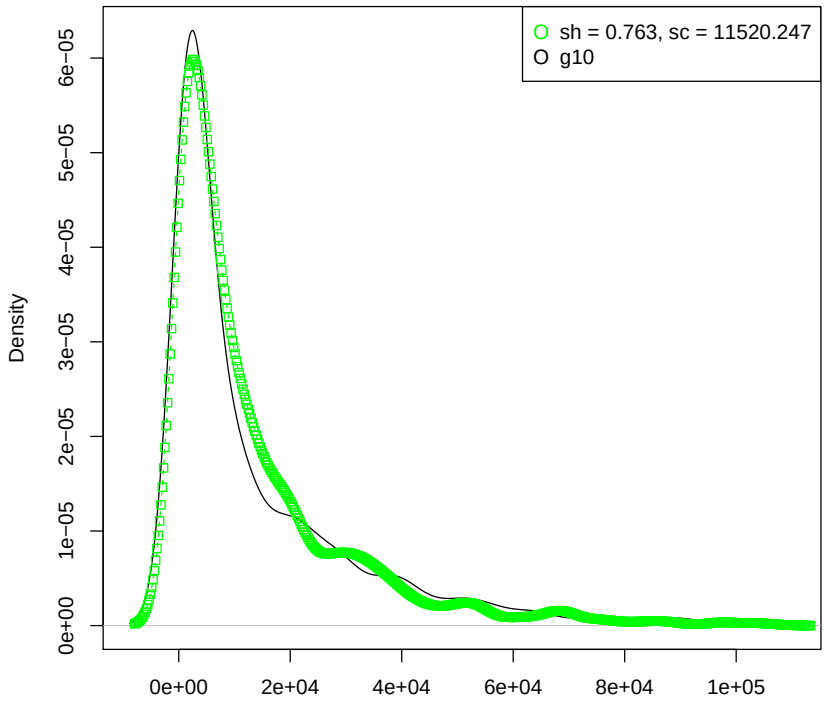

Figure 6.4: PDF: Actual Data & Best-fit Weibull distribution

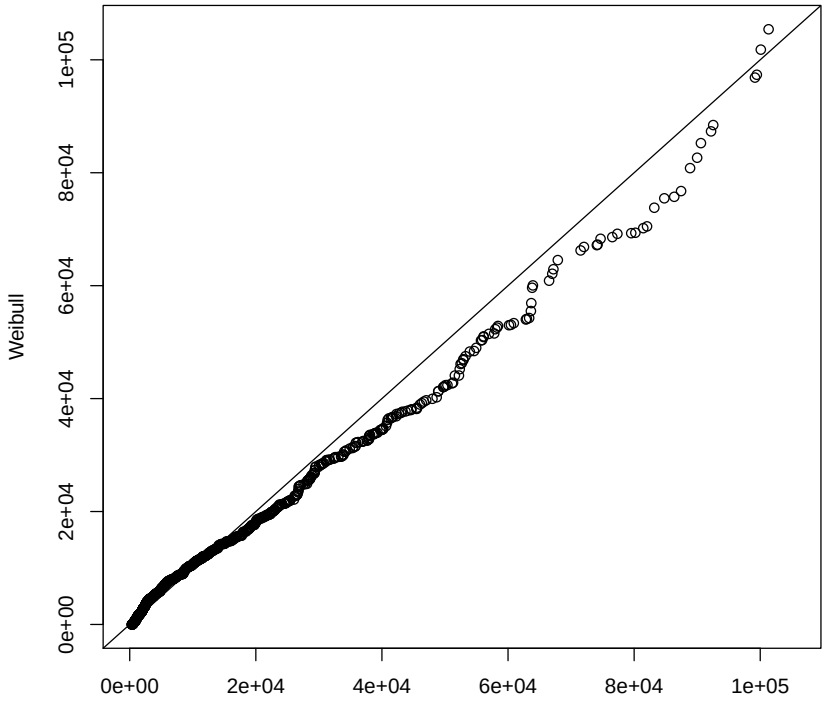

Figure 6.5: Q-Q Plot: Actual Data & Best-fit Weibull distribution

benchmarks from the OR library [151] are considered, which involve scheduling examinations (items) involving specified numbers of students (item sizes), into rooms of specified capacity (bin capacities) within time-slots (number of bins). Two goodness of fit tests are performed on the whole set of instances, to ensure Weibull to be successful in fitting their distribution of item sizes.

The first one is the two-sided Kolmogorov-Smirnov (KS) [6] test. It is a non-parametric test for the equality of continuous, one-dimensional, probability distributions. As implemented in R, it requires two sample sets: One representing the observed data, and the other representing a sample from the hypothesis distribution. The observed data is represented by the item sizes from the real-life instance, while the second set is a vector of items generated according to the best-fit Weibull distribution obtained by MLF from the observed data [158]. The null hypothesis of this statistical test is that the two data sets come from the same underlying distribution. For a 95% level of confidence, if the p-value from the test is at least 0.05, the null hypothesis cannot be rejected.

As a complementary approach the χ^2 test [3] is applied to the instances set. The null hypothesis is that the observed and expected distributions are the same. The procedure requires grouping the items into γ categories, according to their size. Based on these categories, the expected number of values in each category can be computed, assuming that the item sizes are drawn from a Weibull distribution with shape and scale parameters estimated from the data set. Figure 6.6 presents the computation of the χ^2 statistic. On it, the corresponding p-value depends on O and E, where each O_i and E_i represent the observed and expected frequencies of each category i, respectively. This test is known to be less sensitive to outliers in the sample data. For example, referring back to the distribution of Figure 6.4, the tail of the distribution (the values ranging from 20,000 to 100,000, for which the probability is nearly 0) can be grouped into a same category. The other $\gamma - 1$ categories are equally sized.

Table 6.1 presents the results of a randomly selected subset of the analyzed instances. Column `Instance` represents the problem the real-life instances belong to. Whereas the first six rows belong to the ETT, the last six belong to the ROADEF/EURO one. Columns Shape and `Scale` represent the shape and scale parameters of the best-fit Weibull distribution obtained by MLF, respectively. Column p-`value` represents the p-value of the KS test. Last three columns represent the results for the χ^2 test, with number of categories created, the lower bound of the last category (selected by observation in the data of each instance) and the achieved p-value.

$$\chi^2 = \sum_{i=1}^{\gamma} (O_i - E_i)^2 / E_i,$$

Figure 6.6: χ^2 Statistic

Instance	Shape	Scale	p-value	#(cat)	lbTail	p-value
Nott	1.044	43.270	**0.7864**	7	100	**0.059**
MelA	0.946	109.214	**0.091**	10	427	**0.073**
MelB	0.951	117.158	**0.079**	5	47	**0.051**
Cars	1.052	85.438	0.037	18	53	**0.109**
hec	1.139	138.362	**0.436**	10	293	**0.204**
yor	1.421	37.049	**0.062**	7	117	**0.068**
$a1_3^2$	0.447	104,346.70	0.005	30	163,000	**0.105**
$a1_3^3$	0.549	88,267.85	0.001	15	54,800	**0.068**
$a2_1^5$	0.562	67,029.83	0.000	30	470,000	**0.768**
$a2_4^4$	0.334	103,228.30	0.001	30	500,000	**0.051**
b_6^3	0.725	40,469.74	0.000	20	185,000	**0.060**
b_3^5	0.454	91,563.28	0.000	30	140,000	**0.088**

Table 6.1: Statistics Test for the Instances

The results of the KS test reveal that most of the ETT item size distributions can be accurately modeled by a Weibull distribution, since the corresponding p-values are above 5% (highlighted in bold). Unfortunately, these KS results clearly reject the null hypothesis for the ROADEF/EURO instances, most likely due to both the size of the data sets and the presence of outliers in the tail of the distribution. However, it is known that when dealing with large data sets with a small number of large outliers, this test tends to underestimate the p-value. That is, even if the null hypothesis is rejected the candidate distribution might still characterize the data set [64]. This idea is reaffirmed by the results of χ^2, where the null hypothesis cannot be rejected for any of the benchmarks that are presented. Thus, it proves that the Weibull distribution successfully fits the item size distributions of the whole set of proposed real-life instances.

6.2 The Empirical Analysis Layout

This section describes the layout process being performed to carry out the empirical analysis. A generated benchmark suite (based on the Weibull distribution) is used to run the experiments, which include the application of both CP($\mathcal{FD}$) and heuristics techniques. In the CP($\mathcal{FD}$) side, the Gecode constraint solving library is used. It is applied by two equivalent C++ CP($\mathcal{FD}$) Gecode and a CFLP($\mathcal{FD}$) $\mathcal{TOY}(\mathcal{FD}g)$ models. In the heuristics side, the well-known MAXREST, FIRSTFIT, BESTFIT and NEXTFIT BPP strategies are used. They are applied by a C++ model formulating each of these methods.

Figure 6.7 presents the layout process. The first module represents the benchmark generator, and it is described in Section 6.2.1. The benchmark consists of an instance set, each of them contained in a single file (gathering the items sizes of the instance

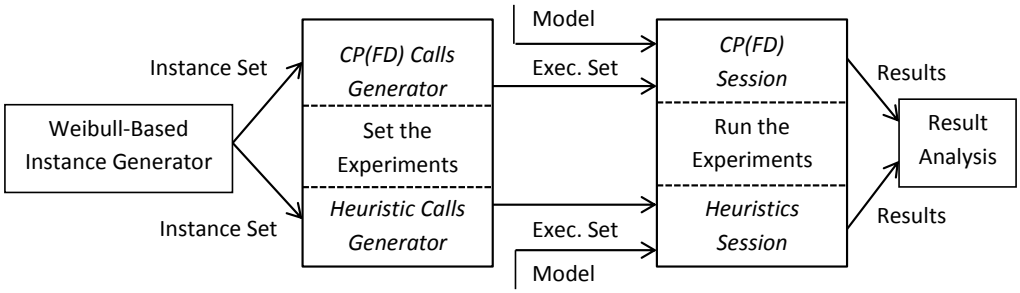

Figure 6.7: The Empirical Analysis Layout

in a decreasing order), which is univocally determined by the *id* of the instance. The second module represents the set up for the experiments, and it is described in Section 6.2.4. It receives the instance set and returns some script files, gathering the set of calls to be performed on the CP($\mathcal{FD}$) or heuristic session (where each call runs a single instance). The CP($\mathcal{FD}$) and heuristics models are described in Section 6.2.2 and 6.2.3, respectively. The third module represents the CP($\mathcal{FD}$) and heuristics sessions, taking both the concrete model and the script to run the instance set. It is also described in Section 6.2.4. Finally, the last module is the analysis of the results, and it is further presented in Section 6.3. The set of instances, the CP($\mathcal{FD}$) and heuristics models and the session scripts are available at: `http://gpd.sip.ucm.es/ncasti/models.zip`.

6.2.1 Instance Set Generation

The benchmark suite is generated by using the Boost library [30]. This is a C++ API that includes type definitions for random number generators and a Weibull distribution, which is parameterized by the random number generator, the shape (k) and the scale (λ). Iteration capabilities for traversing the distribution of generated values are also provided. In the benchmark, λ is fixed to 1000, so that the distributions of item sizes are spanned over three orders of magnitude. For k, a very large range $[0.1, 0.2, \ldots, 19.9]$ of values are considered, giving rise to up to 199 different (k, λ) parameter combinations (or categories). By referring back to Figures 6.2 and 6.3, it is observed that very different item size distributions are generated in such these categories.

With the aim to obtain statistically significant performance figures 100 instances are generated for each category, thus leading to the generation of 19,900 instances. Each instance contains 100 items, which are isolated in a single file (and represented in a decreasing order). The file can be univocally identified by the (k, λ) and the index (0, $\ldots$, 99) within the category. Moreover, the files are enumerated from 1 to 19,900. Thus, for example, the generated 100 instances for $k = 2.5$ are represented in $inst\_2401(2.5, 1000, 0).data$, $\ldots$, $inst\_2500(2.5, 1000, 99).data$. As an example, the instance $inst\_2405(2.5, 1000, 4).data$ contains the following item sizes: $\{2001, 1699, 1657,$

$1647, 1591, 1556, 1534, 1498, 1480, 1466, 1451, 1374, 1365, 1352, 1352, 1350, 1335, 1306, 1298, 1259, 1243, 1224, 1223, 1223, 1212, 1208, 1207, 1202, 1183, 1180, 1175, 1161, 1139, 1133, 1115, 1101, 1093, 1091, 1062, 1062, 1059, 1058, 1005, 981, 979, 970, 969, 955, 946, 941, 928, 923, 916, 911, 888, 854, 849, 844, 809, 808, 808, 803, 769, 753, 728, 716, 672, 670, 665, 656, 651, 622, 591, 588, 570, 567, 558, 554, 552, 538, 527, 527, 507, 503, 490, 450, 437, 402, 386, 371, 365, 355, 325, 321, 312, 297, 205, 193, 177, 135\}$.

Once the instance set is fixed, 11 different scenarios are considered for solving each instance. They consist of setting the size of the bin C to the size of the highest item of the instance times a factor ranging from 1.0 to 2.0 (increasing it 0.1 on each new scenario). Thus, these scenarios (which are not taken from real-life instances) allow to very precisely control how the applied solving methods behave as the size of the bins slightly increase. For example, for solving $inst\_2405(2.5, 1000, 4).data$, in the 11 different experiments C is considered to be 2001, 2202, 2402, 2602, 2802, 3002, 3202, 3402, 3602, 3802 and 4002, respectively.

6.2.2 The CP($\mathcal{FD}$) Model

The CP($\mathcal{FD}$) experiments use the Gecode solver and the most efficient BPP C++ CP($\mathcal{FD}$) model included in the Gecode distribution [173]. Also, for completeness with the topic of this part of the thesis, now a CFLP($\mathcal{FD}$) variant of this BPP model has been implemented in $\mathcal{TOY}(\mathcal{FD}g)$. Regarding the empirical analysis being performed, this $\mathcal{TOY}(\mathcal{FD}g)$ model has been shown to be equivalent to the C++ CP($\mathcal{FD}$) one. That is, both models post the very same $\mathcal{FD}$ constraint network and search strategy to the solver, obtain the very same results for each of the instances of the benchmark and spent the very same time for obtaining such these results. For example, both the Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ models find an optimal solution of 47 bins (in 0.209 seconds) for the instance $inst\_2405(2.5, 1000, 4).data$ proposed before.

The BPP model receives the following input arguments: The id of the instance `Id`. The list of item sizes S. The capacity of the bins C. The timeout used for search exploration T, which is set to 10 seconds to ensure that the session finishes in a reasonable amount of time. It has also been verified that increasing this timeout to 300 seconds does not significantly increase the proportion of solved instances. The model computes as a result if the instance was solved to optimality or not within the used timeout. If so, it also displays the optimal number of bins found and the time spent in search exploration.

The model first employs the L_1 method by Martello and Toth [136] and a first-fit BPP heuristic (which packs each item into the first bin with sufficient capacity), to respectively obtain a lower `lb` and upper bound Ub on the minimum number of bins to be used. Then, it declares the following variables: A variable `Bins` (with initial domain in the range `lb..`ub), representing the number of bins used to pack the items. A vector of 100 variables `Bin` (with initial domain in the range `0..Ub-1`), where each `Bin[i]`

represents the bin the item i is assigned to. A vector of Ub variables Load (with initial domain 0..C), where each Load[i] represents the load of its associated bin. The main constraint is the global BPP constraint proposed by Paul Shaw [177], enforcing that the packing of items into bins corresponds to the load variables. Besides that, three kinds of symmetry breaking constraints are applied: Load ones, stating that, if Bins is smaller than i, then Load[i-1]..Load[Ub-1] are equal to 0. Capacity ones, placing into different bins to those items whose size is greater than half of C. Bin ones, avoiding different solutions involving permutations of items with equal size.

The search strategy labels first the variable Bins. Thus, for each labeled value, the problem is reduced to find out if there is a feasible assignment of the items by using that concrete number of bins. Also, by labeling Bins in an increasing order, the first solution found is known to be optimal. Once labeled Bins, the 100 variables of Bin are labeled in textual order, fitting each item to concrete bins. At this point, and as it happened either with the classical CP($\mathcal{FD}$) problems and with the ETP (cf. Sections 4.3 and 5.4, respectively), the use of an *ad hoc* search strategy highly improves the solving performance for the BPP. That is, for each Bin[i] variable, instead of selecting a naïve increasing value order strategy, the possible bin to place the item is selected by using the *Complete Decreasing Best Fit: (CDBF)* strategy proposed in [82]. It places Bin[i] into the bin b with sufficient but least free space. If the assignment fails, then b (and other bins b1,...,bj with same remaining free space) are pruned from the domain of Bin[i] (and from the domain of other Bin[i1],...,Bin[iz] representing items of the same size).

6.2.3 The Heuristics Models

For the heuristics, a publicly available C++ implementation of the well-know heuristics MaxRest, FirstFit, BestFit and NextFit [159] has been used. These four methods are not exhaustive and thus, given a concrete instance, the optimal number of bins these heuristics compute for it can be worse (higher) than the one computed by the two CP($\mathcal{FD}$) models. For example, for the instance $inst\_2405(2.5, 1000, 4).data$ proposed before, whereas MaxRest uses 48 bins and NextFit 63, both FirstFit and BestFit use 47 bins (as the CP($\mathcal{FD}$) models). On the other hand, being $n = 100$ the number of items, the worst case running time for the MaxRest, FirstFit, BestFit and NextFit heuristics are $\Theta(n\ logn)$, $\Theta(n^2)$, $\Theta(n^2)$ and $\Theta(n)$, respectively. Thus, the experiments will succeed in finding a solution (within the timeout) for the entire 19,900 instance set, as a difference to the CP($\mathcal{FD}$) approach.

Similarly to the CP($\mathcal{FD}$) search approach, the four heuristics place the items in textual order. However, neither of them compute an initial lower nor upper bound in the amount of bins to be used. Instead of that, they start creating a first bin, for which the first item is automatically assigned to. Starting from that the items are traversed to place them in order, creating new bins on demand. In particular, when placing item i

on the set of bins $b_1, \ldots, b_j$ already opened:

- MAXREST traverses $b_1, \ldots, b_j$ in order, computing the bin b_k with maximum remaining space. If it does not fit i, then it opens a new b_{j+1} and assigns i to it.
- FIRSTFIT traverses $b_1, \ldots, b_j$ in order, assigning i to the first bin b_k fitting it. If none of them can do it, then it opens a new bin b_{j+1} and assigns i to it.
- BESTFIT traverses $b_1, \ldots, b_j$ in order, computing the bin b_k with sufficient but least free space for fitting i. If none of them can do it, then it opens a new bin b_{j+1} and assigns i to it. As it can be seen, this behavior is similar to the one of CDBF. However, in the CP($\mathcal{FD}$) approach, for placing each item `Bin[i]`, CDBF acts over the potential set of bins (labeled value of the variable `Bins`), whereas in BESTFIT the candidate set of bins dynamically grows while placing the different items. Moreover, the BESTFIT strategy is not complete, and thus it does not backtrack to place i in a different bin than b_k (as CDBF does).
- Finally, NEXTFIT focuses just on the last bin b_j of the $b_1, \ldots, b_j$ set. If it does not fit i, then it opens a new bin b_{j+1} and assigns i to it.

6.2.4 Setting the Experiments

Once described the instance set and both the CP($\mathcal{FD}$) and heuristics models, this section focuses on the setup for running the experiments. The setup for the Gecode and heuristics C++ models is presented first. Then, the slight differences arisen in $\mathcal{TOY}(\mathcal{FD}g)$ are discussed.

In C++, the models are compiled to generate executables. These executables can be triggered via a command in a command session. The Gecode and heuristics models are adapted so that, instead of S, they receive the name of the instance the items must be read from (e.g., $inst\_2405(2.5, 1000, 4).data$), and, instead of C, they receive the factor to be applied to the first item being read (e.g., 1.0). Also, they receive the name of the file where the solution must be stored (e.g., $inst\_2405(2.5, 1000, 4).data$).
The command `bpp.exe 2405 "$models_path$/bpp_instances/inst_2405(2.5, 1000, 4).data" "$models_path$/Gecode/C++/bpp_solutions/inst_2405(2.5, 1000, 4).sol" 2.5 1.0 10000` solves the instance $inst\_2405(2.5, 1000, 4).data$ using the Gecode model. The generated $.sol$ file contains a single line `ID = 2405 Solved = 1 Time = 218 Bins = 47`, indicating that the instance 2405 has been solved in 218 milliseconds by using an optimal amount of 47 bins.
The command `bpp.exe 2405 "$models_path$/bpp_instances/inst_2405(2.5, 1000, 4).data" "$models_path$/Heuristics/bpp_solutions/inst_2405(2.5, 1000, 4).sol" 1.0` solves the instance $inst\_2405(2.5, 1000, 4).data$ using the heuristics model. The generated $.sol$ file contains four lines, one per heuristic, indicating the instance solved, the time spent and the number of bins used.

```
ID = 2405 MaxRest: 0.000 48
ID = 2405 FirstFit: 0.000 47
ID = 2405 NextFit: 0.000 63
ID = 2405 BestFit: 0.000 47
```

A script (`.bat`) file is generated, containing the 19,900 commands running each of the instances of the benchmark. The execution of this script is called a *session*. As for each instance there are 11 configurations (with C in $[1.0, 1.1, \ldots, 2.0]$) up to 11 script files are used (for example, the file bpp_session_1.0.bat runs the entire instance set, using for each instance bins with the capacity of its highest item).

The execution of a *session* in $\mathcal{TOY}(\mathcal{FD}g)$ requires a few changes w.r.t. the process presented before. First, $\mathcal{TOY}(\mathcal{FD})$ does not compile to native machine language, so each command running an instance is in fact a $\mathcal{TOY}(\mathcal{FD})$ goal (to be executed within a system session). Second, a $\mathcal{TOY}(\mathcal{FD})$ goal cannot read from a file, and thus the goal must include the list of items S as an argument. As it can be seen, the content of the instance set is needed for the creation of the script files, but not further when performing the $\mathcal{TOY}(\mathcal{FD}g)$ *sessions*. Third, the stream of the system can be modified, as to write the solution in a file univocally identified by the id of the instance being run. Thus, the goal bpp 2405 2001 [2001, 1699, 1657, 1647, 1591, 1556, 1534, 1498, 1480, 1466, 1451, 1374, 1365, 1352, 1352, 1350, 1335, 1306, 1298, 1259, 1243, 1224, 1223, 1223, 1212, 1208, 1207, 1202, 1183, 1180, 1175, 1161, 1139, 1133, 1115, 1101, 1093, 1091, 1062, 1062, 1059, 1058, 1005, 981, 979, 970, 969, 955, 946, 941, 928, 923, 916, 911, 888, 854, 849, 844, 809, 808, 808, 803, 769, 753, 728, 716, 672, 670, 665, 656, 651, 622, 591, 588, 570, 567, 558, 554, 552, 538, 527, 527, 507, 503, 490, 450, 437, 402, 386, 371, 365, 355, 325, 321, 312, 297, 205, 193, 177, 135] 10000 `true` == `Result` solves the instance $inst\_2405(2.5, 1000, 4).data$ using the $\mathcal{TOY}(\mathcal{FD}g)$ model. It binds $Result -> 47$ and generates the file $inst\_2405.sol$, which contains a single line `ID = 2405 Solved = 1 Time = 202 Bins = 47` (mate to the one computed by Gecode).

To run the whole instance set, $\mathcal{TOY}(\mathcal{FD})$ cannot make use of scripting, but a non-deterministic function can be used for it. That is, a $\mathcal{TOY}(\mathcal{FD}g)$ script file (e.g., bpp_session_10.toy, which stands for $C = 1.0$) contains just a non-deterministic function bpp_session:: bool. This function receives no arguments, and contains 19,900 conditional rules. Each of them return `true` iff the execution of a concrete instance of the benchmark succeeds (for example, the rule 2405 of bpp_session returns `true` if the goal running the instance 2405 succeeds). The entire benchmark is run by simply including the function bpp_session in the $\mathcal{TOY}(\mathcal{FD}g)$ model and executing the goal bpp_session == true (requesting the system for all the solutions). More specifically, $\mathcal{TOY}(\mathcal{FD})$ suffers memory problems when the function contains more than 300 rules, so to run the whole benchmark the function bpp_session is split in packs of 250 rules and executed in several $\mathcal{TOY}(\mathcal{FD})$ sessions. To automat-

ically trigger the execution of a block of 250 rules, a new function bpp_analysis is created. It is equivalent to the original bpp, but it includes a `fail` primitive just after the `cdbf` search strategy. The underlying idea is that it is the primitive `cdbf` the one reporting the success (and the elapsed time and optimal number of bins) or the fail (and the timeout) to the generated $.sol$ file. Thus, once done that, the execution of the instance goal is no longer important, and it can be explicitly failed with no collateral consequences. By using bpp_analysis in the 19,900 rules of bpp_session, it is for sure that the goal `bpp_session == true` will trigger the execution of the entire benchmark (with no extra interaction of the user with the system).

6.3 Analysis of the Results

Once performed the CP($\mathcal{FD}$) and heuristics sessions, this section analyzes the results. Sections 6.3.1 and 6.3.2 discuss the results of CP($\mathcal{FD}$). As small values of the k parameter ($[0.1, \ldots, 5.0]$) lead to quite different item size distributions, they are first analyzed isolated. Then, the entire k parameter range $[0.1, \ldots, 19.9]$ is considered. The relation among the instance configuration (k, λ and C), the ratio of success of the CP($\mathcal{FD}$) method (within the timeout), and the time and number of bins requested by the solved instances is precisely discussed. Then, Section 6.3.3 presents the results of the heuristics. As they manage to solve the entire instance set (in a nearly negligible amount of time), the discussion is focused on the quality of the solutions for those instances for which CP($\mathcal{FD}$) was successful.

6.3.1 CP($\mathcal{FD}$): Small Weibull k Parameter Values

In this section small k values (ranging in $[0.1, \ldots, 5.0]$ in steps of 0.1) are considered isolated, as they lead to clearly very different regimes in the item sizes distributions (cf. Figure 6.2). Figures 6.8 and 6.9 present the average time required to those instances solved within the timeout, and the percentage of such these instances, respectively. For them, only the capacity factors $C = 1.0$, $C = 1.5$, and $C = 2.0$ are considered. The analysis reveals the following conclusions:

- It is clear that the k factor, which defines the spread of item sizes, has a dramatic impact on the average time taken to find the optimal solution to an instance. The lower values of k correspond to distributions that have greater skew towards smaller items. As k increases, for example consider value 1.5, there is a much greater range of possible item sizes. Then, for even higher values, as 5.0, the distribution of item sizes becomes more symmetric.

  This shift in item size distribution impacts the difficulty of bin packing earlier when the capacity of the bin is smaller. Figure 6.8 presents the effort required when C is equal to the largest item ($C = 1.0$). It can be seen that the range of k

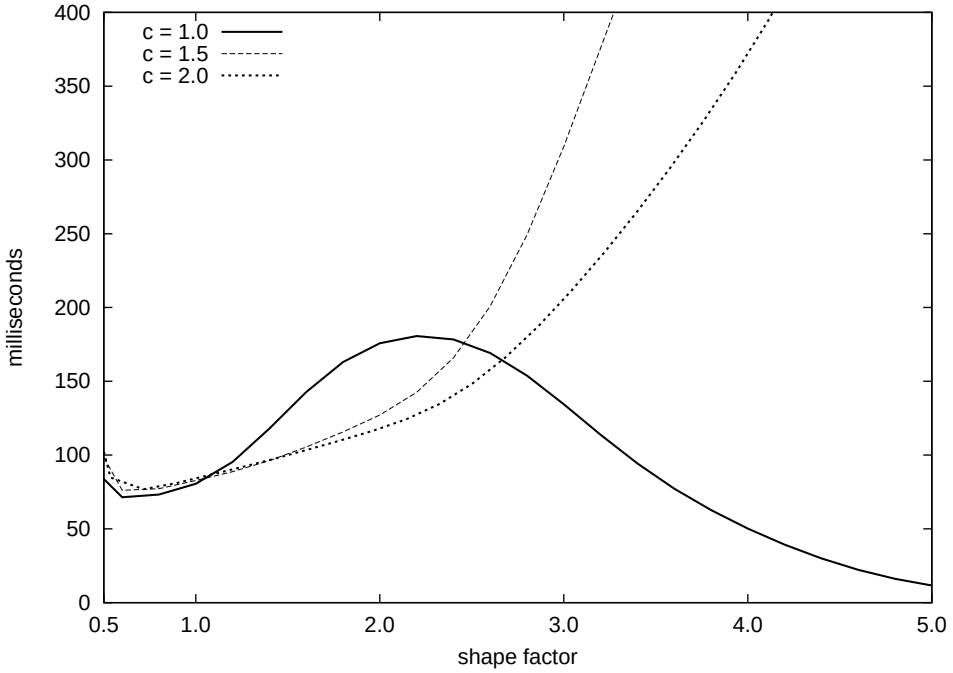

Figure 6.8: Small Shape: Average Runtime for Solved Instances

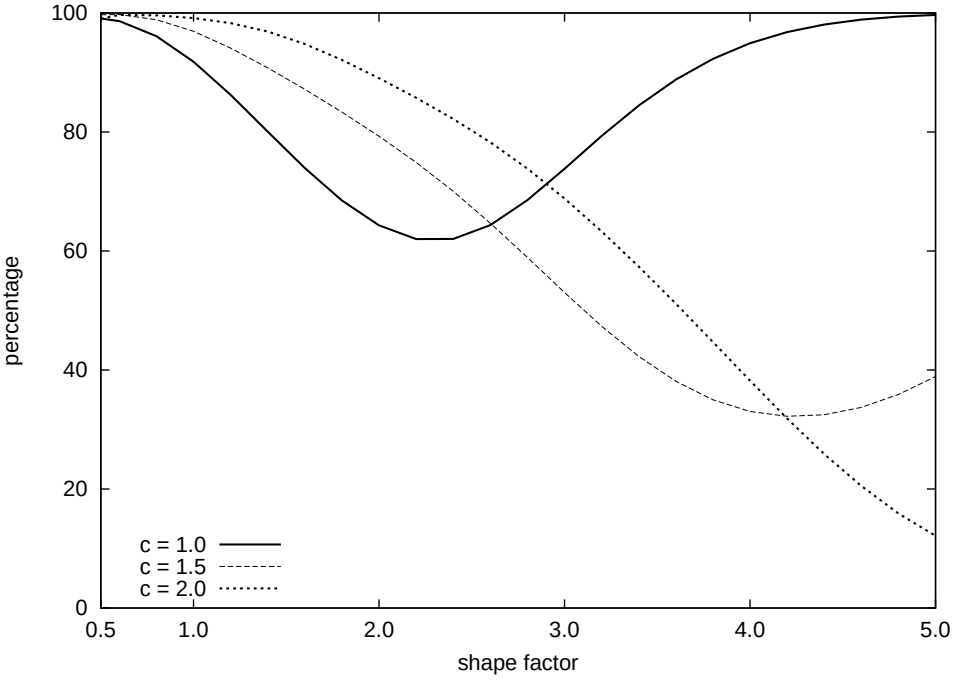

Figure 6.9: Small Shape: Percentage of Solved Instances

for which the instances turn hard for the CP($\mathcal{FD}$) approach is quite narrow. More specifically, for the range $k = [1.2, 1.3, \ldots, 3.2]$ the mean of the elapsed time for those instances that did not timeout is above 100 milliseconds (with a maximum of 175ms for $k = 2.2$). This reveals a combination of k and C leading to a challenging combination of items to be placed in the same bin.

As the k parameter increases above 3.2 the item size distributions become more *compact*, i.e., with a smaller range of items, also excluding very small and very large items. In this setting, with C remaining fixed to the highest item, the combination of k and C becomes again less challenging, and the instances are solved in less than 100ms.

- This challenging combination of k and C is also reflected in Figure 6.9, where the percentage of instances being solved is presented. It can be seen that for this very same range of $k = [1.2, 1.3, \ldots, 3.2]$ the percentage of instances solved is below the 80%, reaching a minimum for $k = 2.2$ with just a 60% of the instances being solved. This makes an remarkable split of the instances: Whereas a 60% of them are solved in about 175ms, the other 40% are not solved in 10,000ms. That is, for this particular combination of k, C and search strategy, very slight deviations on the sizes of the items represent the difference between solving the instance in a nearly negligible amount of time or not being able to solve it.

- Fixing the range of k and increasing C the easy-hard-easy pattern seems to happen again. For example, Figure 6.9 shows that, with $C = 1.5$ there is also a range of $k's$ for which the problem turns hard. However, now this range starts at $k = 2.0$, which is higher than the $k = 1.2$ of $C = 1.0$.

  Also, the range is wider, as it reaches a minimum at $k = 4.2$, but for $k = 5.0$ it is still below the 80% of solved instances. Moreover, the computational challenge of these $k = [2.0, \ldots]$ and $C = 1.5$ combinations is harder than the one of $k = [1.2, 1.3, \ldots, 3.2]$ and $C = 1.0$. In the minimum of $k = 4.2$ (which again, as in $C = 1.0$, holds both for percentage and running time), only a 30% of the instances are solved, and the mean of their running time is of 550ms. Both easy-hard-easy patterns of $C = 1.0$ and $C = 1.5$ share a same upper bound of nearly a 100% of instances solved for $k = [0.1, \ldots, 0.5]$. But, as it can be seen, increasing C also increases the computational challenge of the problem. Next section will describe it further by considering the whole range of k.

6.3.2 CP($\mathcal{FD}$): Full Range of k Parameters

The Weibull distributions generated with a high k value differ from those with the smaller values studied above (cf. Figure 6.3). Essentially, these distributions have lower spread shown by successively taller density functions centering towards the value of the λ parameter. In this section, the entire range of $k = [0.1, \ldots, 19.9]$ is considered,

allowing to observe the easy-hard-easy pattern for each of the 11 configurations of C. Figures 6.10 and 6.11 present both the average running time of the instances solved within the timeout and the percentage of instances that this corresponds to. Now, also the average number of bins associated with these instances is considered, as it is presented in Figure 6.12. All the instances contain 100 items, and thus the average number of items placed per bin can be extracted as well from the results of Figure 6.12.

As mentioned before, increasing C also increases the computational challenge of the problem, and thus more instances reach the timeout when being solved with the CP($\mathcal{FD}$) method. The effectiveness of the CP($\mathcal{FD}$) method is measured, classifying each concrete category (k,λ) of 100 instances by the percentage of instances being solved: Ranging in 80%-100%, in 60%-80%, in 40%-60%, in 20%-40% or in 0%-20%. Thus, and considering again just $C = 1.0$, $C = 1.5$ and $C = 2.0$, Figure 6.11 shows that, whereas $C = 1.0$ contains just categories of 80%-100% and 60%-80%, $C = 1.5$ contains categories of 80%-100%, 60%-80%, 40%-60% and 20%-40%, and finally $C = 2.0$ contains categories of 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%.

Table 6.2 summarizes the results from figures. Columns C and k-Range represent the selected bin capacity and the range of shapes, respectively. For each concrete C, the groups are presented in increasing k order, with a (*) identifying the concrete k for which the percentage of instances being solved is minimum. Focusing on each concrete row, columns `Percentage` and `Time` represent the group it belongs to and the range of average running times achieved by it, respectively. Then, columns `I/B` and `Bins` represent the range of average number of items per bin and the range of number of bins achieved. Finally, columns `MaxRest` and `NextFit` (which are discussed further in Section 6.3.3) represent the range of average deviations of the heuristics MAXREST and NEXTFIT w.r.t. the optimal number of bins found by CP($\mathcal{FD}$), respectively. The results of Table 6.2 reveal the following conclusions:

- The amount of instances being solved decreases as C increases. In $C = 1.0$, 178 categories belong to group 80%-100%, i.e., 17,800 instances of the benchmark belong to categories (k,λ) for which CDBF is able to solve between 80%-100% of them. Then, 21 categories belong to group 60%-80%, i.e., the other 2,100 instances of the benchmark belong to categories (k,λ) for which CDBF is able to solve between 60%-80% of them. In $C = 1.5$, 150 categories belong to group 80%-100%, 17 to group 60%-80%, 18 to group 40%-60% and 14 to group 20%-40%. In $C = 2.0$, 87 categories belong to group 80%-100%, 20 to group 60%-80%, 12 to group 40%-60%, 18 to group 20%-40% and 62 to group 0%-20%.

  As it can be seen, in all the C configurations, the big mass of instances belong to group 80%-100%, and then there is a more or less fair division among the other groups. However, in $C = 2.0$ this does no longer hold, as there are 62 categories for which CDBF is able to solve at most a 20% of them, and for 12 of these categories the percentage reaches nearly a 0%.

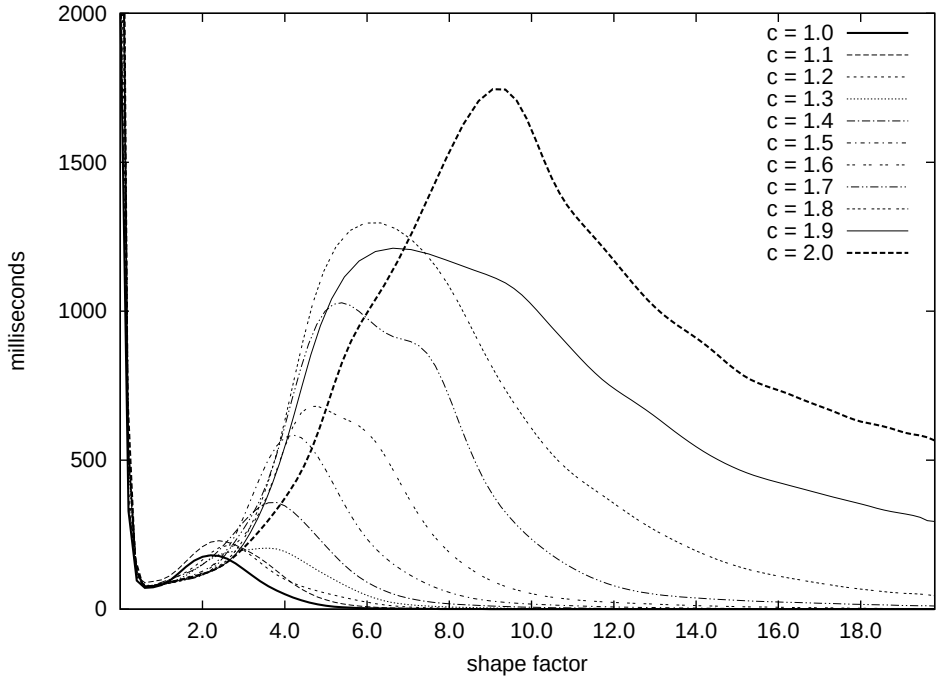

Figure 6.10: Average Runtime for Solved Instances

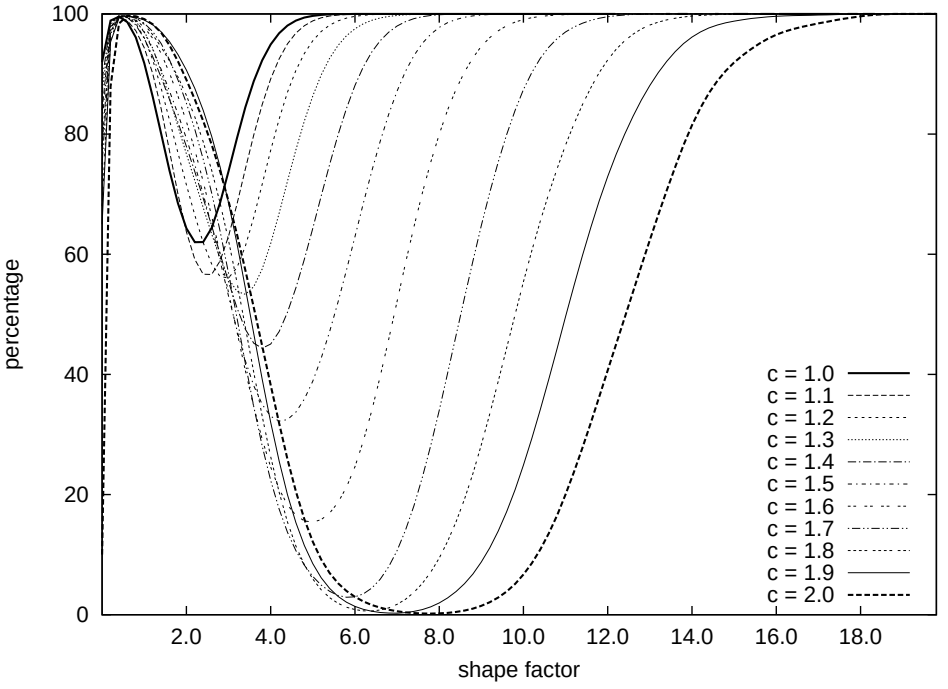

Figure 6.11: Percentage of Solved Instances

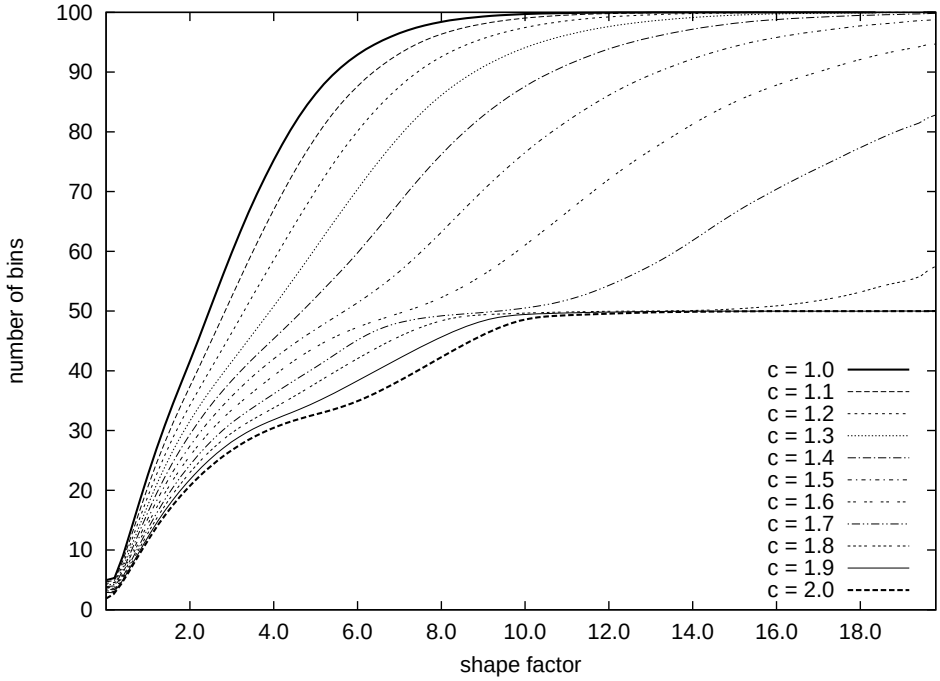

Figure 6.12: Number of Bins

- As mentioned in Section 6.3.1, there is a correlation between the percentage of instances solved and the time CDBF devotes to search exploration. Focusing on column `Time` of Table 6.2, it can be seen that the categories belonging to group 80%-100% are the fastest in being solved. In $C = 1.0$, this time ranges in 50-100ms and 0-100ms. In $C = 1.5$, this time ranges in 50-150ms and 0-150ms. However, in $C = 2.0$, whereas for the range $k = [0.1, \ldots, 2.8]$ the average time ranges in 50-200ms, for the range $k = [14.1, \ldots, 19.9]$ it ranges in 550-1000ms, reaching effort levels of categories of groups 20%-40% and even 0%-20%.

  In general, it can be seen that the average effort time for categories belonging to the same group are nearly equivalent, with an slightly increase as C increases. For example, categories of group 60%-80% spend around 100-175ms if $C = 1.0$, around 150-250ms if $C = 1.5$, and around 200-250ms if $C = 2.0$ (with, once again, the remarkable exception of range $k = [12.8, \ldots, 14.0]$, which spends around 850-1000ms).

- From Figure 6.12 it is clear that, as k increases, the amount of bins used in the optimal solutions increases as well. In $C = 1.0$, it can be seen that $k = 10.0$ represents a frontier, as from this value on all the instances are solved by using 100 bins (i.e., placing each item into a different bin). Also, referring back to Figure 6.10 it can be seen that the effort for solving such these instances is nearly 0ms.

C	k-Range	Percentage	Time	I/B	Bins	MaxRest	NextFit
1.0	0.1-1.1	80-100	50-100	3.70-100	1-27	0.00-0.15	0.0-5.0
	1.2-3.2	60-80	100-175	1.61-3.57	28-62	0.15-0.25	5.0-14.5
	(*) 2.2	60	175	2.22	45	0.35	14.5
	3.3-19.9	80-100	0-100	1.00-1.59	63-100	0.00-0.15	0.0-12.5
1.5	0.1-1.9	80-100	50-150	3.84-100	1-26	0.00-0.21	0.0-6.0
	2.0-2.7	60-80	150-250	3.22-3.70	27-31	0.21-0.30	6.1-9.0
	2.8-3.8	40-60	250-400	2.70-3.13	32-37	0.31-0.61	9.1-12.5
	3.9-5.2	30-40	400-550	2.17-2.63	38-46	0.53-0.64	12.5-19.0
	(*) 4.2	30	550	2.38	42	0.64	15.80
	5.3-5.9	40-60	250-400	2.08-2.13	47-48	0.23-0.52	19.1-20.1
	6.0-6.8	60-80	150-250	1.82-2.04	49-55	0.05-0.22	19.5-20.0
	6.9-19.9	80-100	0-150	1.00-1.79	56-100	0.00-0.05	0.0-19.5
2.0	0.1-2.8	80-100	50-200	4.00-100	1-25	0.00-0.23	0.0-3.5
	2.9-3.5	60-80	200-250	3.70-3.85	26-27	0.24-0.38	3.6-4.9
	3.6-4.0	40-60	250-400	3.44-3.57	28-29	0.39-0.58	5.0-5.2
	4.1-4.8	20-40	400-550	3.33	30	0.59-0.80	5.2-5.6
	4.9-7.0	0-20	550-1150	2.70-3.23	31-37	0.80-1.40	5.1-5.6
	(*) 7.1-8.2	1-2	1250-1550	2.38-2.63	38-42	1.05-1.40	2.5-5.1
	8.3-11.0	0-20	1250-1750	2.00-2.33	43-50	0.15-1.05	0.5-2.5
	11.1-12.0	20-40	1150-1250	2.00	50	0.13-0.15	0.5
	12.1-12.7	40-60	1000-1150	2.00	50	0.08-0.12	0.3-0.5
	12.8-14.0	60-80	850-1000	2.00	50	0.05-0.08	0.0-0.3
	14.1-19.9	80-100	550-1000	2.00	50	0.00-0.05	0.0

Table 6.2: Summary of the Results for $C = 1.0$, $C = 1.5$ and $C = 2.0$

Thus, it can be concluded that the instances become trivial, as the CP($\mathcal{FD}$) model takes advantage of the capacity symmetry breaking constraints (cf. Section 6.2.2) to solve the instance by using just the initial constraint propagation (with no CDBF search exploration being performed).

In the case of $C = 2.0$, obviously $k = 10.0$ represents also a frontier, as from this value on all the instances are solved by using 50 bins (i.e., as the bins double their capacity, then two items are placed on each of them). However, by referring back to Figures 6.10 and 6.11 it can be seen that, for these range $k = [10.0, \ldots, 19.9]$, the problem is still computational challenging. That is, the computational challenge of deciding which two items are placed on each bin lead to categories of groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%, with an average search effort time ranging in 550-1600ms for them.

For $C = 1.5$, this computational challenge also makes the number of bins used in the optimal solution to keep growing once reached $k = 10.0$ (until reaching a 100 bins solutions with $k = 19.9$).

- As the amount of bins used in the optimal solutions increases as k increases, it is clear that there are concrete intervals of items per bins for the categories belonging to groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%. However,

these intervals slightly change for the different C values.

Regarding categories classified into group 80%-100%: The interval is $1.00 - 1.59$ and $3.70 - 100.00$ for $C = 1.0$. It is $1.00 - 1.79$ and $3.84 - 100.00$ for $C = 1.5$.Finally, it is 2.00 and $4.00 - 50.00$ for $C = 2.0$. Thus, it can be concluded that, if the category requires, for its 100 instances, an average of less that 1.60 or more than 4.00 items per bin, then the category is going to be classified into group 80%-100%.

Unfortunately, for the rest of groups there are no general conclusions for the three $C = 1.0$, $C = 1.5$ and $C = 2.0$ configurations. For example, for group 60%-80%: The interval is $1.61 - 3.57$ for $C = 1.0$. It is $1.82 - 2.04$ and $3.22 - 3.70$ for $C = 1.5$ Finally, it is 2.00 and $3.70 - 3.85$ for $C = 2.0$. Thus, there is no single value of items per bin that belongs to the three C configurations. Same situations happens for groups 40%-60% and 20%-40%, for which only $C = 1.5$ and $C = 2.0$ provide categories.

Last but not least, in $C = 2.0$ and $k = [10.0, \ldots, 19.9]$ there are categories of groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%, all of them with a same amount of 2.0 items per bin.

In summary, it can be seen that increasing the capacity C dramatically increases the computational difficulty of the problem, as shown by both the amount of instances solved and the search time spent for the solved ones. However, no general conclusions are obtained about the relation between the average items placed per bin and the computational challenge of the problem.

6.3.3 Heuristics: Quality of the Obtained Solutions

The four heuristics MAXREST, FIRSTFIT, BESTFIT and NEXTFIT solve the entire 19,900 instance set for the 11 C configurations in a nearly negligible amount of time. However, once again, they are not exhaustive methods, and thus the solutions they found can be worse than the one found by the CP($\mathcal{FD}$) approach, in the sense of containing a higher amount of bins. This section revisits Table 6.2, analyzing the results of columns `MaxRest` and `NextFit`, which represent the range of average deviations of MAXREST and NEXTFIT w.r.t. the optimal number of bins found by CP($\mathcal{FD}$), respectively. The difference in performance between MAXREST, FIRSTFIT, and BESTFIT is very small, so only the first one is presented. On each case, the instances for which CP($\mathcal{FD}$) was not successful are left out of the heuristics analysis.

Figures 6.13 and 6.14 present the results for MAXREST and NEXTFIT, respectively. As it can be seen, the former behaves around an order of magnitude better than the latter, with the differences with the CP($\mathcal{FD}$) approach ranging in 0.0-1.4 bins and 0.0-20.0 bins, respectively. Furthermore, the results of Table 6.2 reveal the following conclusions:

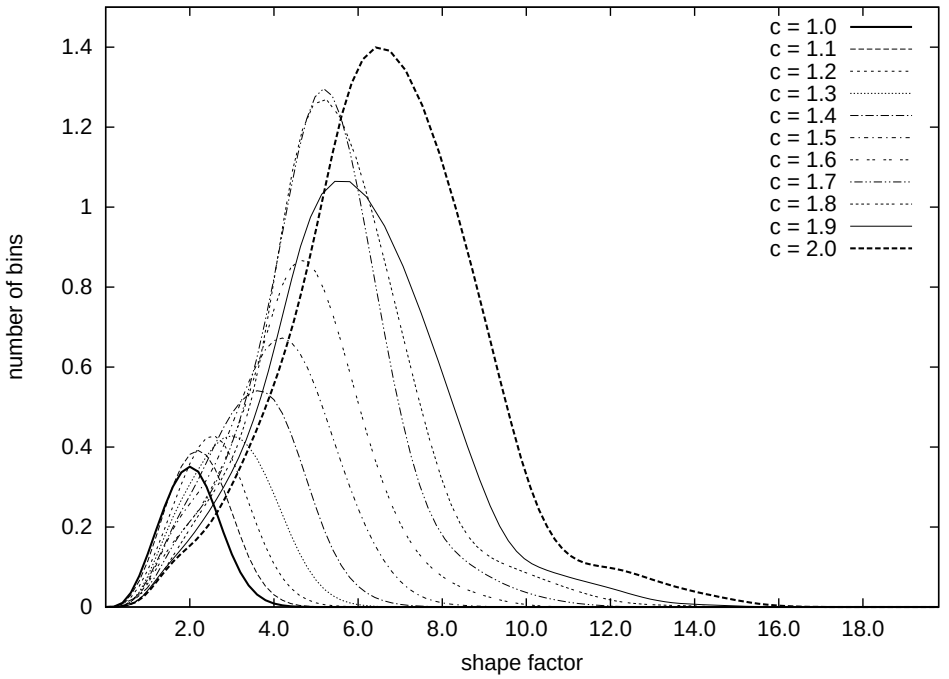

Figure 6.13: MaxRest: Difference in Average Number of Bins

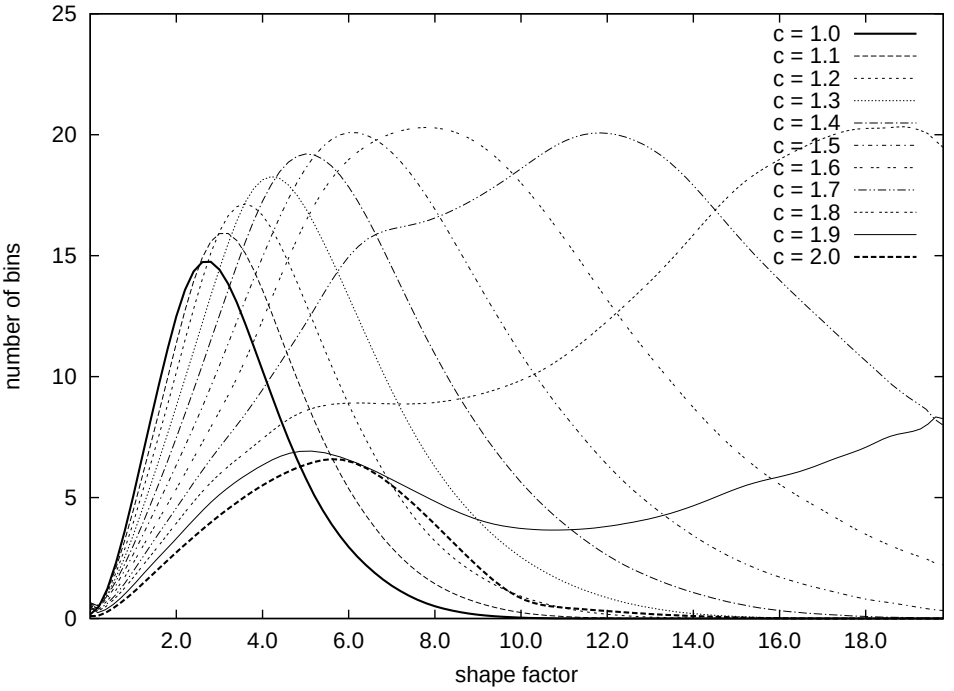

Figure 6.14: NextFit: Difference in Average Number of Bins

- In MAXREST, it is clear that, as C increases, the average gap achieved by the heuristic w.r.t. the CP($\mathcal{FD}$) optimal solutions increases as well. This is intuitive, since increasing the computational challenge of the problem makes it harder to give a good quality solution. For $C = 1.0$, the gap ranges in $0.00 - 0.35$ bins, for $C = 1.5$ it ranges in $0.00 - 0.64$ and for $C = 2.0$ it ranges in $0.00 - 1.40$.

  Moreover, for each concrete C, there is a clear correlation between the categories classified by the CP($\mathcal{FD}$) approach (groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%) and the average gap achieved by the heuristic. That is, the harder the group is for CP($\mathcal{FD}$) the higher the gap achieved. In $C = 2.0$, there is again a frontier, this time at $k = 11.0$. It makes that, for $k = [11.1, \ldots, 19.9]$, the average gap for categories 80%-100%, 60%-80%, 40%-60% and 20%-40% are all in the range $0.00 - 0.13$.

  Thus, it is concluded that the use of the heuristic MAXREST (and thus also of FIRSTFIT and BESTFIT) represents a very good alternative to the CP($\mathcal{FD}$) approach, as it achieves very good solutions in a nearly negligible amount of time. This is specially remarkable in the case of $C = 2.0$. On it, for the range $k = [4.9, \ldots, 11.0]$ the categories belonged to group 0%-20% (moreover, for the range $k = [7.1, \ldots, 8.2]$ the percentage of instances being solved was nearly a 0%) and requested an average search effort ranging in 550-1650ms. Now, by using the heuristic, they are all solved in a nearly negligible amount of time, and with an average gap ranging $0.15 - 1.40$ bins.

- In NEXTFIT, there is a frontier in $C = 1.5$. From $C = 1.0$ to $C = 1.5$, as C increases, the average gap achieved by the heuristic w.r.t. the CP($\mathcal{FD}$) optimal solutions increases as well. For $C = 1.0$, the gap ranges in $0.00 - 14.50$ bins, and for $C = 1.5$ it ranges in $0.00 - 15.80$. But then, from $C = 1.6$ to $C = 2.0$ the achieved curve representing the gap changes, and it gets smaller w.r.t. the CP($\mathcal{FD}$) optimal solutions. For $C = 2.0$, the gap ranges in $0.00 - 5.60$ bins, which is not just even better than $C = 1.5$, but also than $C = 1.0$.

  Moreover, there is no correlation between the categories classified by the CP($\mathcal{FD}$) approach (groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%) and the average gap achieved by the heuristic (for neither $C = 1.0$ nor $C = 1.5$ nor $C = 2.0$). For example, in $C = 1.0$, both the ranges $k = [0.1, \ldots, 1.1]$ and $k = [3.3, \ldots, 19.9]$ lead to categories classified into group 80%-100%. However, whereas in the former the gap ranges in $0.00 - 5.00$ bins, in the latter it ranges in $0.00 - 14.50$.

  In $C = 1.5$, something similar happens, with both the ranges $k = [0.1, \ldots, 1.9]$ and $k = [6.9, \ldots, 19.9]$ leading to categories classified into group 80%-100%. However, whereas in the former the gap ranges in $0.00 - 6.00$ bins, in the latter it ranges in $0.00 - 19.50$, (where this 19.5 represents nearly the maximum gap achieved with the heuristic). In $C = 2.0$, it also happens. But, here, the issue of the

frontier at $k = 11.0$ (that happened in MAXREST) still holds. For $k = [11.1, \ldots, 19.9]$, the average gap for categories 80%-100%, 60%-80%, 40%-60% and 20%-40% are all in the range $0.00 - 0.50$ (which is nearly the minimum gap achieved by the heuristic).

Thus, it is concluded that the heuristic NEXTFIT is not competitive w.r.t. MAXREST (and thus also of FIRSTFIT and BESTFIT) and the CP($\mathcal{FD}$) approach. But, for $C = 2.0$ and the range $k = [11.1, \ldots, 19.9]$ it is competitive w.r.t. all of them.

6.4 Related Work

As mentioned before, the BPP is a ubiquitous problem that arises in many practical applications. Amongst the many applications of this problem are scheduling, stock cutting, television commercial break scheduling, and container packing [42, 61]. It is closely related to a variety of other problems such as rectangle packing [181, 180, 195, 67]. Recent work has focused on geometric generalizations of bin packing [43]. Typical BPP methods rely on either Mathematical Programming [42], Satisfiability techniques [86], CP($\mathcal{FD}$) [166, 177] and heuristics [17, 70]. There are many known bounds on the optimal number of bins which can be used in most of the techniques mentioned above [61, 116, 135].

6.5 Conclusions

In this chapter an empirical analysis of the solving hardness of the classical BPP has been presented, as the basis for the future development of portfolio solvers, which will tackle generalized BPP real-life problems coming from the data centre optimization industry. The analysis includes the application of both CP($\mathcal{FD}$) and heuristic techniques to the solving of a parametric generated benchmark of BPP instances, and a detailed analysis of the results.

Regarding the BPP instances used in the analysis, a new benchmark suite based on the well known Weibull distribution has been presented. The flexibility of the Weibull approach has been discussed, showing that a great variety of item size distributions can be generated via different combinations of the (k,λ) parameters. A set of real-life BPP instances (coming from the data centre optimization and education industries) have been gathered, showing that the Weibull approach allows to model them very accurately. To visually show the quality of these fits, MLF and Q-Q plots have been used. Besides that, a more rigorous analysis has been done by applying the Kolmogorov-Smirnov and χ^2 statistical tests.

The layout process being performed to carry out the empirical analysis has been presented. A benchmark suite of 19,900 instances has been generated, consisting of

199 different categories (of 100 instances of 100 items each) or combinations of (k,λ). Specifically, whereas λ has been fixed to 1000 (spanning the item sizes of the distributions over three orders of magnitude), the k has ranged in $k = [0.1, 0.2, \ldots, 19.9]$. Once the instance set is fixed, 11 different bin capacities C have been tried, setting it to the size of the highest item of the instance times a factor ranging from 1.0 to 2.0 (increasing it 0.1 on each new scenario).

Then, the CP($\mathcal{FD}$) and heuristics models have been presented. In the CP($\mathcal{FD}$) side, two equivalent C++ CP($\mathcal{FD}$) Gecode and CFLP($\mathcal{FD}$) $\mathcal{TOY}(\mathcal{FD}g)$ models are applied. In the heuristics side, a publicly available C++ implementation of the well-know heuristics MAXREST, FIRSTFIT, BESTFIT and NEXTFIT has been applied. Also, the setup for running the experiments has been discussed, analyzing the C++ scripts generated for Gecode and the heuristics and their adaptation to the $\mathcal{TOY}(\mathcal{FD}g)$ framework.

The analysis of the CP($\mathcal{FD}$) results have revealed that, for each of the 11 values of C being considered, there are categories (k,λ) for which the CP($\mathcal{FD}$) approach finds it hard to solve the whole instance set of 100 instances. Moreover, the hardness a concrete category results for the CP($\mathcal{FD}$) method can be classified by using 5 groups: 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%. Thus, it can be seen that increasing C also increases the computational challenge of the problem, so that the amount of instances being solved decrease as C increases. In $C = 1.0$, 178 categories belong to group 80%-100% and 21 to group 60%-80%. In $C = 1.5$, 150 categories belong to group 80%-100%, 17 to group 60%-80%, 18 to group 40%-60% and 14 to group 20%-40%. In $C = 2.0$, 87 categories belong to group 80%-100%, 20 to group 60%-80%, 12 to group 40%-60%, 18 to group 20%-40% and 62 to group 0%-20% (the latter with up to 12 categories for which the percentage of instances being solved reached nearly a 0%). Also, there is a correlation between the percentage of instances solved and the time devoted to search exploration.

Also, as k increases, the amount of bins used in the optimal solutions increases as well. As all the instances contain 100 items, there are concrete intervals of items per bins for the categories belonging to groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%. However, these intervals slightly change for the different C values. On the one hand, it can be concluded that, if the category requires an average of less that 1.60 or more than 4.00 items per bin, then the category is going to be classified into group 80%-100%. On the other hand, for the rest of groups there are no general conclusions for the different C configurations. Specifically, for $C = 2.0$ and $k = [10.0, \ldots, 19.9]$ there are categories of groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%, all of them with a same amount of 2.0 items per bin.

The heuristics solve the entire benchmark in a negligible amount of time, and thus the analysis has focused on the average deviations of MAXREST (with similar results to FIRSTFIT, and BESTFIT) and NEXTFIT w.r.t. the optimal number of bins found by CP($\mathcal{FD}$). The deviations of the former are an order of magnitude better (ranging in 0.0-1.4 bins) than the ones of the latter (ranging in 0.0-20.0 bins). In MAXREST, it is clear that, as

C increases, the average gap achieved by the heuristic w.r.t. the CP($\mathcal{FD}$) optimal solutions increases as well. Moreover, there is a clear correlation between the categories belonging to groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20% and the gap achieved by the heuristic. In NextFit, there is a frontier in $C = 1.5$. Thus, whereas from $C = 1.0$ to $C = 1.5$ the gap increases, from $C = 1.6$ to $C = 2.0$ it decreases. Unfortunately, there is no correlation between the categories and the gap achieved. In general, whereas the use of the heuristic MaxRest (and thus also of FirstFit and BestFit) represents a very good alternative to the CP($\mathcal{FD}$) approach, as it achieves very good solutions in a nearly negligible amount of time, the gap achieved by NextFit makes it not that much interesting.

Part IV

Positioning $\mathcal{TOY}(\mathcal{FD})$ w.r.t. Other CP($\mathcal{FD}$) Systems

The third research part of the thesis is focused in positioning $\mathcal{TOY}(\mathcal{FD})$ w.r.t. other state-of-the-art CP($\mathcal{FD}$) systems. Chapter 7 describes an in-depth modeling comparison of the Golomb and ETP problems between the algebraic CP($\mathcal{FD}$) systems MiniZinc and ILOG OPL, the C++ CP($\mathcal{FD}$) systems Gecode and ILOG Solver, the CLP($\mathcal{FD}$) systems SICStus Prolog and SWI-Prolog, and the CFLP($\mathcal{FD}$) systems PAKCS and $\mathcal{TOY}(\mathcal{FD})$. Chapter 8 performs an in-depth solving comparison of the same problems and systems (in the case of $\mathcal{TOY}(\mathcal{FD})$, the three systems versions $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}s)$ are considered).

Chapter 7

Modeling Analysis

CP($\mathcal{FD}$) is widely used nowadays, with a big and alive community building up a large number of applications, including both the modeling of new problems and system implementations. As it was pointed out in the motivation of this thesis, by focusing in the main CP($\mathcal{FD}$) related conferences and journals, it can be seen that there are multiple real-life applications based on algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$) approaches, but there is no single application based on CFLP($\mathcal{FD}$). This reveals that, although the CFLP($\mathcal{FD}$) features make it a suitable paradigm for tackling CSP's and COP's, it is still far away from being taken into account by the CP($\mathcal{FD}$) community. To this end, this chapter positions the system $\mathcal{TOY}(\mathcal{FD})$ w.r.t. other state-of-the-art CP($\mathcal{FD}$) systems. The main purpose of the chapter is to show that $\mathcal{TOY}(\mathcal{FD})$ is appealing w.r.t. any of them for modeling different COP's, thus encouraging its use (and also the use of the CFLP($\mathcal{FD}$) paradigm itself).

For the modeling analysis, both the classical CP($\mathcal{FD}$) problem of the Golomb Rulers (cf. Section 2.1) and the real-life Employee Timetabling Problem (cf. Section 5.3) are used. On the one hand, Golomb is a pure basic CP($\mathcal{FD}$) problem, allowing to discuss the basic concepts of modeling any COP. On the other hand, the ETP is a real-life application, fully parametric, non-monolithic and including CP($\mathcal{FD}$) independent components. Thus, it fully exploits the expressive power of the different paradigms, allowing to analyze in detail the strengths and drawbacks of each of them.

To widen the analysis, two different state-of-the-art systems are chosen per paradigm: MiniZinc and ILOG OPL (for algebraic CP($\mathcal{FD}$)), Gecode and ILOG Solver (for C++ CP($\mathcal{FD}$)), SICStus and SWI-Prolog (for CLP($\mathcal{FD}$)) and, finally, PAKCS and $\mathcal{TOY}(\mathcal{FD})$ (for CFLP($\mathcal{FD}$)).

The chapter is organized as follows: Section 7.1 uses Golomb to provide general insights about the abstraction of the constraint solver, the specification of the $\mathcal{FD}$ variables, $\mathcal{FD}$ constraints and search strategies, as well as the output of the solutions. Then, the next sections use the ETP to discuss in detail how each paradigm tackles a concrete modeling issue: Section 7.2 analyzes the coordination of the ETP stages. Sections 7.3, 7.4 and 7.5 focus on the data structures, variables and constraints, respectively.

Section 7.6 provides a final expressiveness comparison. Each section includes code examples to clarify the concepts being discussed, and the generated models for all the systems are available at: `http://gpd.sip.ucm.es/ncasti/models.zip`. To end this chapter, Section 7.7 compares the lines of code each system needs for modeling both Golomb and ETP. Section 7.8 presents some related work, and Section 7.9 reports conclusions.

7.1 General Modeling Insights

The Golomb problem is used to discuss basic modeling concepts about the constraint solver, $\mathcal{FD}$ variables, $\mathcal{FD}$ constraints, search strategies, and displaying the solutions. Code examples are presented in Figures 7.1 (for C++ CP($\mathcal{FD}$) systems) and 7.2 (for algebraic CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems). Each concept is presented in a separate section, but all of them refer back to the already mentioned figures.

7.1.1 Golomb: Solver Abstraction

C++ CP($\mathcal{FD}$) systems require solver-targeted models. Programming directly using the API of the CP($\mathcal{FD}$) library being used also implies explicit handling of a constraint solver object, and managing the control of its decision variables, constraints, objective function, constraint store, constraint propagation, search engine and search control (as well as the garbage collection of all these elements). Figure 7.1 shows this management in Gecode and ILOG Solver, respectively.

More specifically, as pointed out in Section 2.3.2, a Gecode `Space` object contains the variables, propagators (implementations of constraints), branchers (describing the shape of the search tree) and cost function specification. The constraint solver is made explicit in the Gecode model: First, in line 2, by the use of `Golomb` class (inheriting from `Space`). Then, in line 6, by the definition of the `Golomb` constructor (in charge of specifying the Golomb problem). Finally, in line 20, with the explicit creation of the `solver` object in the `main` program. From now on, in C++ CP($\mathcal{FD}$) systems, the distinction between an object and a class will be omitted when it is clear from the context.

In ILOG Solver, an `IloSolver` object represents the constraint solver (line 34). It contains the variables, constraints, labelings and cost function. Nevertheless, as pointed out in Section 2.3.2, an `IloModel` object (line 33), representing the constraint store, is also needed. Its use replicates information, as any generic `IloIntVar` and `IloConstraint` the `IloModel` contains is different from the targeted `IlcIntVar` and `IlcConstraint` the `IloSolver` deals with. Thus, a dedicated translation from `IloModel` to `IloSolver`, via the `extract` method (line 37), is needed. Finally, the

```
(01) ----------------------------------- GECODE -----------------------------------
(02) class Golomb : public Space {          | Golomb(bool share, Golomb& s) :
(03) protected:                             |  Space(share, s) {
(04)   IntVarArray m;                       |   m.update(*this, share, s.m);
(05)   ...                                  |   ...
(06) public:                                |  }
(07)   Golomb(int n, bool p) : m(*this,n,   |
(08)    0, (int) ceil(pow(2.0,n-1)-1.0)){   | void constrain(const Space& _s) {
(09)     bool sat = true;                   |   const Golomb& s =
(10)     ...                                |    static_cast<const Golomb&>(_s);
(11)     for (int i = 0; i < (n-1); i++){   |   IntVar v(s.m[s.m.size()-1]);
(12)       if (sat){                        |   rel(*this, this->m[this->m.size()
(13)         rel(*this,m[i],IRT_LE,m[i+1]); |                      -1], IRT_LE,v);
(14)         if (p){                        | }
(15)           this->status();              |};
(16)           sat = !(this->failed());     |
(17)         }                              |
(18)       }                                |int main(int argc, char* argv[]) {
(19)     }                                  |   ...
(20)     ...                                |   Golomb* solver = new Golomb(n);
(21)     branch(*this, m, INT_VAR_NONE,     |   BAB<Golomb> engine(solver);
(22)                           INT_VAL_MIN); |   while (Golomb* s = engine.next()){
(23) }                                      |     solver = s;
(24)                                        |   }
(25)   void print() const {                 |   solver->print();
(26)    std::cout << "m=" << m << std::endl;|   ...
(27)   }                                    |}
(28) ---------------------------------- ILOG SOLVER ----------------------------------
(29) int main(int argc, char* argv[]) {     | ...
(30)   ...                                  | IloGoal st = IloGenerate(env, m,
(31)   bool sat = true;                     |          IloChooseFirstUnboundInt);
(32)   IloEnv env;                          | IloGoal g = IloSelectSearch(env, st,
(33)   IloModel model(env);                 |      IloMinimizeVar(env, m[n-1], 1));
(34)   IloSolver solver(model);             | IloNodeEvaluator D_Node =
(35)   IloIntVarArray m(env, n, 0,          |               IloDFSEvaluator(env);
(36)    (int) ceil(pow(2.0,n-1)-1.0));      | IloGoal goal= IloApply(env,g,D_Node);
(37)   ...                                  | solver.extract(model);
(38)   for (int i = 0; i < (n-1); i++){     | solver.startNewSearch(goal);
(39)     if (sat){                          | if (solver.next())
(40)       IloConstraint ct = m[i] < m[i+1];|   for (int i = 0; i < n; i++)
(41)       model.add(ct);                   |     std::cout<< solver.getValue(m[i]);
(42)       if (p)                           | solver.endSearch();
(43)         sat = solver.propagate(ct);    | env.end();
(44)     }                                  | ...
(45)   }                                    |}
```

Figure 7.1: C++ CP($\mathcal{FD}$): Golomb Structure

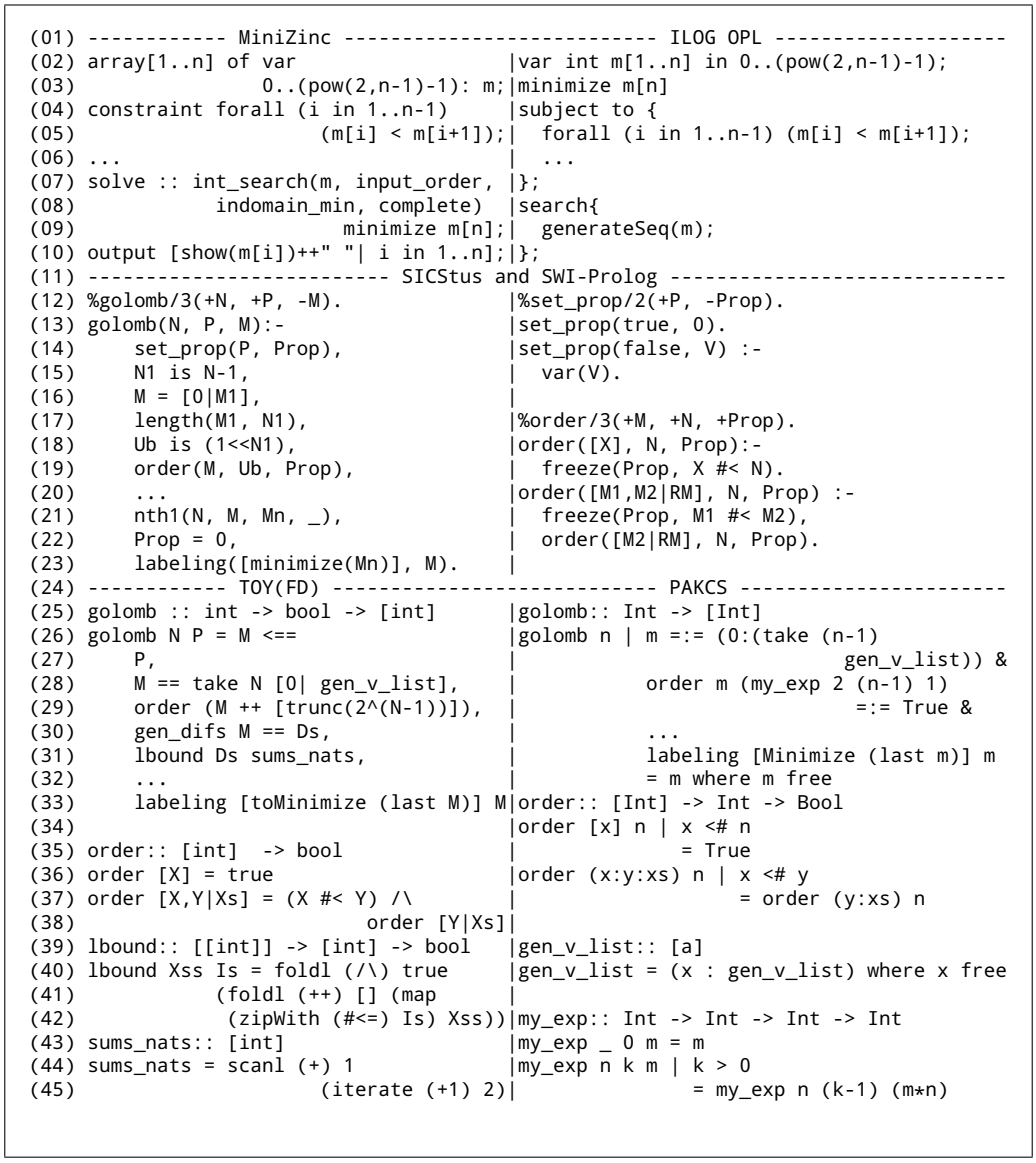

Figure 7.2: Algebraic CP($\mathcal{FD}$), CLP($\mathcal{FD}$), CFLP($\mathcal{FD}$): Golomb Structure

IloEnv object (line 32) manages the memory of any object of the application, providing an end( ) method (line 43), to ease the garbage collection of the application objects.

In the case of algebraic CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems, the constraint solver and its management are transparent to the user.

7.1.2 Golomb: Variables

Each paradigm provides different kind of variables. MiniZinc and ILOG OPL provide decision ($\mathcal{FD}$) variables and parameter variables (which can be seen as constants). Gecode and ILOG Solver inherit the concept of variable from OO programming (typed and mutable). Regarding $\mathcal{FD}$ variables, Gecode supports both IntVarArray variable arrays or IntVarArgs argument arrays. The former must have a fixed length at its declaration, and it uses memory from the home Space. The latter is suitable for temporary variable arrays, those ones either being dynamically built or used as arguments for post functions. CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems coordinate constraint solving with their operational semantics based on SLD resolution and lazy narrowing, respectively. These systems rely on logical variables, which are syntactically unified by a Herbrand ($\mathcal{H}$) solver. The $\mathcal{FD}$ solver is used to manage any $\mathcal{FD}$ primitive constraint arisen in the computation, and each logic variable involved on this primitive must be firstly attributed [105], turning them into an $\mathcal{FD}$ variable (in SICStus, SWI, PAKCS and $\mathcal{TOY}(\mathcal{FD})$ models, this attribution is transparent to the user).

For example, focusing on the m variables, whereas algebraic and C++ CP($\mathcal{FD}$) systems create a fixed-length $\mathcal{FD}$ variable array, the CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) ones create a list formed by an explicit 0 followed by $n-1$ new fresh logic variables. Algebraic and C++ CP($\mathcal{FD}$) systems take advantage from dealing with $\mathcal{FD}$ variables, using lower and upper bounds (0 and $(2^{N-1})-1$, respectively) to set their initial domain. CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems take advantage of pattern matching to implicitly unify m_0 and 0.

As C++ CP($\mathcal{FD}$) systems explicitly handle a constraint solver, the declaration of any variable is attached to it. In Gecode, m is declared as an attribute of the Golomb class, and further initialized in the constructor class. In ILOG Solver, it is attached to the IloModel object, and further dumped to IloSolver. In the rest of the systems, as they abstract the notion of constraint solver, variables are freely declared.

7.1.3 Golomb: Constraints

The ordering of the m variables (i.e., each $m_i < m_{i+1}$) has been selected to present how the different systems declare and propagate the $\mathcal{FD}$ constraints.

In algebraic CP($\mathcal{FD}$) systems, the built-in function forall takes a one-dimensional array and aggregate the elements. MiniZinc and ILOG OPL use it to declare all the constraints in just one line of code. Both systems identify this code as an $\mathcal{FD}$ constraint declaration: MiniZinc by using the reserved word constraint (Figure 7.2, line 4) and ILOG OPL placing the expression within the body of subject to{...}. Regarding the

constraint propagation, both MiniZinc and ILOG OPL only support batch mode, which is set transparently to the user.

In C++ CP($\mathcal{FD}$) systems, Gecode and ILOG Solver use a `for` loop ranging in $0 \ldots n-2$ (Figure 7.1, line 11), constraining on each step the *i*-th variable to be smaller than the next one. Regarding constraint declaration (specify the $\mathcal{FD}$ constraint and post it to the constraint store), Gecode uses the instruction `rel(*this,m[i],IRT_LE,m[i+1])`, declaring `m[i]` to be smaller than `m[i+1]` and posting it to the implicit constraint store of the `Golomb` constraint solver object. In the case of ILOG Solver, it declares the `IloConstraint ct` and then uses the `add` method to post `ct` to the `IloModel` constraint store.

Regarding constraint propagation, Gecode and ILOG Solver receive P as an input argument (Figure 7.1, line 7), setting the Boolean variable `p` to `true` (incremental mode) or `false` (batch mode). They also use the Boolean variable `sat` to control if the constraint store is currently satisfiable or not (line 9). In this setting, if `p`, then the constraints being posted are propagated. Gecode uses the `Space status()` method (line 15) and ILOG Solver the `IloSolver propagate(IloConstraint)` one (note that is the `IloModel` representation the one used as argument). Triggering the constraint store propagation can detect an unfeasible constraint network. Gecode does that by using the `Space failed()` method (line 16), and ILOG Solver by checking the result reported by the `propagate` one. Finally, if the constraint store is not feasible, it makes no sense to continue posting new constraints, so the current status is stored in the variable `sat` and it is used to control the posting of new constraints.

In CLP($\mathcal{FD}$) systems, the predicate `order` (Figure 7.2, line 17) recursively constrains the order of the m variables. Whereas the first clause is the base case (constraining the last variable to be smaller than the `Ub`) the second clause is the recursive case (constraining the *i*-th variable to be smaller than the next one). The expression `M1 #< M2` specifies the constraint and posts it to the constraint store.

Regarding constraint propagation, CLP($\mathcal{FD}$) systems implicitly use incremental propagation mode, but batch mode can be simulated via `freeze(+Flag, +Expr)` predicates, delaying the execution of `Expr` as long as its `Flag` is not ground. The predicate `set_prop` is placed at the beginning of the `Golomb` predicate (line 14), setting the propagation mode to be applied to the whole program. It receives P as its input parameter and returns `Prop` as a result. Whereas the first clause considers the incremental mode (returning the Prolog atom `0`) the second clause considers the batch mode (returning the new fresh logic variable `V`). Either `0` or `V` is passed as the flag to all the program predicates posting constraints (as, for example, in line 19). Orthogonally, if batch mode is used, all the constraints must be posted to the constraint store and being propagated before the search strategy specified in `labeling` starts. Thus, `Prop` and `0` are unified just before declaring the search (line 22). Whereas in incremental mode it makes no effect (as `Prop` was already unified to `0` before) in the batch one it triggers in order the posting and propagation of all the delayed constraints.

In CFLP($\mathcal{FD}$) systems, the function `order` (line 33) is mate to the predicate `order` of the CLP($\mathcal{FD}$) systems, but the argument `Prop` is omitted. In $\mathcal{TOY}(\mathcal{FD})$, the primitives `batch_off` and `batch_on` (cf. Section 3.2.5) set the propagation mode to incremental and batch, respectively. Thus, different propagation modes can be applied to different program fragments. In this case, it is sufficient to set the concrete propagation primitive at the very beginning of the program (line 27). In PAKCS, only incremental propagation is supported.

7.1.4 Golomb: Search Strategy Declaration

As a COP, an exhaustive search on the m variables is needed to find a solution, specifying the variable set, the variable order, the value order and the cost function, as well as a search control. Whereas algebraic CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems provide good abstractions for most of these concepts, their management in Gecode and ILOG Solver becomes harder.

In Gecode, as pointed out in Section 2.3.2, the shape of the search tree to be explored is specified via `Branchers`. The instruction `branch`, placed at the end of the `Golomb` constructor (Figure 7.1, line 21), declares `m` to be the variable set to be labeled, following a left to right order and an increasing domain value order for each variable.

Then, a `Search Engine` object `BAB<Golomb> engine` (line 21) is in charge of search control. It receives the `Golomb` solver previously created to perform a branch and bound exploration of its search tree, where each node of the tree is a `Space`. The `next()` method of the engine (line 22) triggers the search, returning a pointer to the `Golomb Space` placed in the solution node. As exploring the tree is based on cloning `Spaces`, the `Golomb` copy constructor (line 2) is needed to specify how to perform a clone. It relies on the method `update` (line 4), which allows to clone the `m` variables (also including the propagators posted on them). Thus, updating all the variables of a `Space` suffices to clone it.

Regarding the BAB exploration, Gecode allows to modify the shape of the tree during exploration, by adding new constraints. The method `constrain` (line 8) receives a current solution node (`Space& _s`), explores the value of one of its variables (in this case m_{n-1}: `IntVar v(s.m[s.m.size() -1])`) and forces the rest of the tree exploration to obtain better lower bounds for this variable (by posting `rel(*this, this-> m[this-> m.size()-1], IRT_LE,v)`). Gecode specifies a cost function 'expr' by setting it to an optimization `var`, which is forced to be minimized/maximized in the `constrain` method of the `Space` model class. Then, search is just controlled via a `while` loop (line 22), which computes the solutions one by one (forcing each new solution to have a better bound for m_{n-1} than the older one). Once the loop finds no more solutions, the last solution is the optimal one (line 23).

In ILOG Solver, search strategies are performed via the execution of `IloGoals`, daemons attached to an `IloSolver` object, and executing an algorithm on it (lines 36

and 38). It is said that an `IloGoal` fails if its attached `IloSolver` becomes inconsistent by running its execution method; otherwise the goal succeeds (line 39). The library provides predefined `IloGoals` to abstract the different elements of a search declaration. Thus, the goal `st` (line 30) plays the role of the Gecode `Brancher`. Under the env context, it uses the predefined `IloGenerate` to specify the labeling of `m` in textual order and increasing value order. The goal g (line 32) plays the role of the Gecode `constrain` method, with the third parameter requiring the bound of each new m_{n-1} found to be 1 value smaller than the former one. The node evaluator `D_Node` sets a depth first search exploration of the tree. The combination of g and `D_Node` allow to create the goal `goal` (line 36), to be applied to the constraint solver (line 38). The search control is then performed via the methods `startNewSearch(goal)`, `next()` and `endSearch()`.

The former approach was presented due to its similarity with the Gecode one. However, ILOG Solver allows an easier approach, using an `IloObjective` (in this case `IloObjective obj = IloMinimize(env, m[n-1]);`), which is added to `IloModel` and specifies the cost function. With it, an `IloGoal` as the one of line 30 can be directly applied to `IloSolver` (skipping thus the `IloGoals` of line 32 and 36). Anyway, as a goal is an algorithm to be applied over an `IloSolver`, in both approaches a solution is thus the current state of `solver` after running the algorithm.

The rest of algebraic CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems abstract the notion of search control to the user, who thus cannot customize it. Regarding search declaration, MiniZinc specifies a search strategy via a search annotation within the `solve` block of the program (Figure 7.2, line 7). It specifies a `complete` search for an $\mathcal{FD}$ problem (`int_search`), labeling `m` in `input_order` and in a increasing value ordering (`indomain_min`). Finally, the cost function is specified with `minimize m[n]`. In the case of ILOG OPL, it wraps in a `search` block the primitive `generateSeq` (line 9), using `m` as the variable set, and also uses `minimize m[n]` (line 3) to specify that it is an $\mathcal{FD}$ optimization problem, wrapping the constraints in a `subject to` block (line 4). In SICStus, SWI, PAKCS and $\mathcal{TOY}(\mathcal{FD})$, the `labeling` expression acts over `m`, and the cost function is specified via `minimize Mn` (lines 23, 31 and 33, respectively).

7.1.5 Golomb: Showing the Solution

Once a solution for m has been computed, it has to be displayed to the user. The systems use different ways to access the values of the variables. In Gecode and ILOG Solver, `m` variables are explicitly attached to `solver`, and it must be used in order to look for their solutions. The former contains `m` as a `protected` attribute of the `Golomb` class (Figure 7.1, line 4), and thus the devoted access method `print()` is used (line 28). The latter makes use of the predefined `getValue(IloIntVar)` method (line 41). Note again that the ILOG Concert constraint store object is used as a parameter accessing to its mate ILOG Solver constraint solver one. In MiniZinc, ILOG OPL, SICStus,

SWI, PAKCS and $\mathcal{TOY}(\mathcal{FD})$, the variables are not attached to a solver, and thus they can be directly accessed. All the systems but MiniZinc automatically display m to the user by default. In the case of SICStus, SWI, PAKCS and $\mathcal{TOY}(\mathcal{FD})$, this happens because `M` is an output parameter in the `golomb` main predicate/function (Figure 7.2, lines 12 and 25, resp). In the case of ILOG OPL, because all the variables are automatically output as a result. Finally, MiniZinc makes use of a predefined `show` method within the `output` block of the program (line 10).

7.2 ETP: Solver Abstraction

The ETP is used to discuss the implementation of the p_tt algorithm on each paradigm. It was presented in Section 5.3, and is referred back several times from now on. The algorithm relies on a four stage process: `team_assign` (to find tda bijections), `tt_split` (to generate the independent nt subproblems), `tt_solve` (to sequentially solve them) and `tt_map` (to map the solution to $timetabling$).

To coordinate the different stages of p_tt, a search for a feasible tda_i must be done in `p_tt_ta_7`. Then, using this tda_i, stages `tt_split`, `tt_solve` and `tt_map` are triggered, obtaining (in `p_tt_so_8`) a suboptimal $timetabling_i, eh_i$. The execution then backtracks to `p_tt_ta_7`, looking for a new tda_j, and repeating the process. Once no more tda are found, the suboptimal pairs $timetabling_k, eh_k$ are compared, and the one with smallest eh is outputted as the final result. An important remark is that, whereas $Table$ is created in $team\_assign$ (`p_tt_ta_1`), its content is split into $tt_1, \ldots, tt_{nt}$ in tt_split (`p_tt_sp_4`).

7.2.1 Algebraic CP($\mathcal{FD}$)

Figure 7.3 presents the ETP program structure in MiniZinc (the ILOG OPL one has been omitted as it is quite similar). A different model (`*.mzn` file) is used to implement each stage (lines 2 and 11), with its input arguments initialized in a `*.dzn` file. Thus, an external script is required to coordinate the execution of the different files. Programming the script represents an additional difficulty on itself, as it is a task independent from the modeling of a problem in MiniZinc. Coordinating the execution layout of the models requires using the information displayed by the `output` block of each `*.mzn` stage, generating with it the input arguments of the `*.dzn` file of next stage (e.g., `d`, `e` and `o_abs` between `team_assign.mzn` and `tt_split.dzn`). In tt_solve, the script must generate `nt` independent `tt_solve.dzn` (one per tt_i) input files, to be run over the `tt_solve.mzn`. Then, their generated `output` must be gathered for the `tt_map.dzn` input. To manage all these files, the script must avoid a name clash. Although ILOG OPL provides a native scripting language, in this modeling comparison the coordination of the ETP instances of the benchmark is done by hand.

```
(01) ---------------------------------- MiniZinc -----------------------------------
(02) --------- Team_Assign.mzn -----------|----------- TT_Split.mzn ----------------
(03) int nD; int nT; int nTW; ...         |array[1..nDays] of 1..nTeams: d; ...
(04) array[1..nDays] of var 1..nT: d; ... |array[1..nTeams, 1..days_per_team,
(05) constraint alldifferent([d[i] |      | 1..(nTW+1)] of var 0..24: totZ;
(06)                    i in 1..nT]); ... |constraint forall (i in 1..nD,j in 1..nTW)
(07) solve :: int_search(d, input_order,  | (tZ_aux[i,j] = 1-(sum (z in 1..(nT*nTW)
(08)      indomain_min, complete) satisfy;| where o_abs[i,z]=((d[i]-1)*nTW+j)) (1)));
(09) output [show(d[i]) ++ " " |          |solve satisfy;
(10)               i in 1..nDays ] ++ ...;|output ["h = ", show(h), "\n"] ++ ...;
(11) --------- TT_Solve.mzn --------------|----------- TT_Map.mzn ------------------
(12) array[1..nD,1..(nTW+1)] of 0..1:totZ;|array[1..nT] of 0..10000: eh;
(13) array[1..nD, 1..(nTW+1)]             |array[1..nD] of 1..nT: d;
(14)                      of var 0..24: tt;|array[1..nD, 1..((nT*nTW)+1)] of var
(15) constraint forall (i in 1..nD,       |                  0..24: timetabling;
(16)         j in 1..(nTW+1)) (tt[i,j] in |constraint forall (i in 1..nD, j in 1..
(17)         slots_per_jour[dC_list[i]]); | ((d[i]-1)*nTW)) (timetabling[i,j]=0);
(18) solve :: int_search([tt[i,j] | i in  |solve satisfy;
(19)  1..nD, j in 1..(nTW+1)],input_order,|output ["hp=",show(hp),"\n"] ++ ... ;
(20)  indomain_min, complete) minimize eh;|
(21) output ["eh=",show(eh),"\n"] ++ ... ;|
```

Figure 7.3: Algebraic CP($\mathcal{FD}$): ETP Program Structure

The use of four models (files) is mandatory, as `team_assign.mzn` must be isolated from `tt_split.mzn`, and the solving of each tt_i requires a dedicated execution of `tt_solve.mzn`. Regarding the former, in a `*.mzn` model each input argument can be either a parameter or a decision variable, but not both. Variable d must act as a decision variable in `team_assign.mzn` (line 4), as `p_tt_ta_4` sets its initial domain to `1..nt`. But, it must act as a parameter variable in `tt_split` (line 3), as `p_tt_sp_2` uses it to compute $TotZ$. Lines 4-8 show the computation of the values for `tZ_aux` (a Boolean decision variable array used to further compute `totZ`). On them, the content of `d[i]` is placed in a `where` condition (filtering the `z` values being considered to compute `tZ_aux[i,j]`), and these condition expressions only support parameter variables.

Isolating `team_assign` does not prevent from generating a two model (files) approach, where `tt_split`, `tt_solve` and `tt_map` are gathered in the second one. This model is feasible, and a MiniZinc implementation of it is included in the models provided. However, this approach does not exploit the independence of the $tt_1, \ldots, tt_{nt}$ generated subproblems, leading to $eh_1, \ldots, eh_{nt}$ suboptimal solutions (optimal in the context of each subproblem), and thus ensuring the final eh (equal to the sum of these $eh_1, \ldots, eh_{nt}$) to be optimal. This is due to the fact that each `*.mzn` model must include a single `solve` declaration. Thus, if a two models approach is used, the variables of $tt_1, \ldots, tt_{nt}$ must be gathered in the variable set of this single `solve` declaration. By selecting, for example, a lexicographic variable selection order, then the optimal combination of values for variables of tt_1 is not computed just once, but one time per each

feasible combination of values of $tt_2, \ldots, tt_{nt}$. This makes the approach absolutely inefficient (e.g., for the ETP-15 instance of Section 5.4, whereas the four models approach finds the optimal solution in seconds, the two models approach does not find it after several hours).

Finally, whereas the native scripting language of ILOG OPL would make possible to fit the coordination of the different stages proposed before, the lack of such a native scripting language in MiniZinc precludes it from doing so. The Gecode/FlatZinc front-end `fzn-Gecode` [172] includes a parameter `-n` to specify the number of solutions (0 to obtain all of them), but it does not include a parameter to simulate the method `SearchEngine::next()` (computing one tda_i per each iteration of the stages). Also, an `output` block cannot display an unbound $\mathcal{FD}$ variable (as it is the case of $Table$ in `team_assign.mzn` and of tt in `tt_split.mzn`). Thus, the decision made has been to skip creating $Table$ and create tt from scratch in tt_solve. To still take advantage of the binding to 0 of some tt variables (due to absences in o_abs), an array of Boolean `0..1` variables tt_info is created in tt_split. Thus, in `p_tt_so_3` $totZ$ is traversed, binding to 0 those variables of tt_info representing absences, but also binding to 1 the rest of them (having all its variables bound, tt_info can be displayed in the `output` of `tt_split.mzn`). Then, `tt_solve.mzn` creates tt following the dimensions of tt_info, and uses its content to do the corresponding bindings.

7.2.2 C++ CP($\mathcal{FD}$)

Figure 7.4 presents the ETP program `main` entry point in Gecode (the ILOG Solver one has been omitted as it is quite similar). The model is contained in just one file. Whereas the outermost `while` loop of the `main` method (line 10) coordinates the execution of the stages, the innermost loop (line 2) ensures the sequential solving of $tt_1, \ldots, tt_{nt}$.

The independent CP($\mathcal{FD}$) stages `tt_split` and `tt_map` are easily implemented by using the C++ abstractions. The stage `team_assign` requires a constraint solver `s` (line 5), for finding the different feasible tda. For each tda_i, the stage `tt_solve` creates a new vector `slv` (line 14), with each `slv[i]` dedicated to the solving of tt_i. In ILOG Solver, each solver is represented as an `IloSolver` (with its associated `IloModel`), but in Gecode `s` and `slv` lead to the different `Space` subclasses `StageI` and `StageIII` (taking advantage of the Gecode capability to abstract each independent CP($\mathcal{FD}$) problem under a different class). Anyway, using a different solver entity for finding tda and solving a tt_i is mandatory, as the API of CP($\mathcal{FD}$) systems preclude the posting of new constraints on a solver entity being in search mode (which is the case of `s` when reaching stage tt_solve). In the case of Gecode, even the use of different `StageI` and `StageIII` classes is mandatory, as if both were merged, then the `copy()` method would crash on trying to clone any $\mathcal{FD}$ variable not even being initialized.

The association of variables to solvers forces them to be passed as arguments to any program point where the solutions of the variables are required. This is the case

```
(01) ---------------------------------- Gecode ------------------------------------
(02) int main(int argc, char* argv[]) {   |   while ((sat) && (i < nT)){
(03)  ... //parse input args              |    sat = slv[i]->tt_solve(nTW, ws, eF,
(04)  //I. Team Assign                    |              T, h, p, W, SS, dc_list);
(05)  StageI* s = new StageI;             |    if (sat){
(06)  bool sat = s->team_assign(nD, nT,   |     BAB<StageIII> eng(slv[i]);
(07)     nTW, eR, ws, abs, dC, p, oAbs);  |     while (StageIII* ss= eng.next())
(08)  if (sat){                           |      slv[i] = ss;
(09)   DFS<StageI> engine(s);             |     eh += slv[i]->get_eh();
(10)   while (StageI* sol= engine.next()){|    }
(11)    //II. TT Split                    |    i++;
(12)    sat = true;                       |   }
(13)    s = sol;                          |   //IV. TT Map
(14)    vector<StageIII*> slv;            |   if (sat)
(15)    vector< vector<int> > dC_list;    |    tt_map(nT, nTW, eh, s, slv);
(16)    int h = tt_split(nD, nT, nTW, ws, |  }
(17)           oAbs, dC, s, slv, dC_list);| }
(18)    //III. TT Solve                   |... //clean elements
(19)    int eh, i = 0;                    |}
```

Figure 7.4: Gecode ETP Program `main` Entry Point

of `s` (in charge of labeling `d`), which is passed as an argument to the (even CP($\mathcal{FD}$) independent) stages `tt_split` (line 16) and `tt_map` (line 15), just to give them access to the computed values for `d`. Finally, the isolation from `s` and `slv` makes impossible to fit the bindings between $Table$ and $tt_1, \ldots, tt_{nt}$ of the coordination of p_tt stages proposed before, as the mate variables to be linked by equality constraints are posted to different solvers. Thus, $Table$ is skipped and tt is created from scratch (via `slv`) in tt_solve. To still take advantage of the binding to 0 of some tt variables (due to absences in o_abs), `slv` is also passed to `tt_split` (line 16), where $totZ$ is used to constrain to 0 the corresponding variables.

7.2.3 CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$)

Figure 7.5 presents the `p_tt` main program predicate (line 1) in SICStus (respectively main program function (line 12) in $\mathcal{TOY}(\mathcal{FD})$). SWI-Prolog and PAKCS have been omitted as they are quite similar. The model is contained in just one file, and the code is simpler and neater than the one for algebraic CP($\mathcal{FD}$) and C++ CP($\mathcal{FD}$) systems, with a declarative coordination of the stages by simply placing them in order.

The use of multiple solvers, the assignment of constraints to them and their coordination is abstracted to the user, which just has to focus on declaring the different $\mathcal{FD}$ constraints arisen on `team_assign` and `tt_solve`. Regarding the $\mathcal{FD}$ variables which, once again, are freely declared, a set of reflection functions allow to access their domain during the goal computation. When this domain is reduced to a singleton, the variable is automatically unified to the single value in the domain. In particular, `D` va-

```
(01) ---------------------------------- SICStus ------------------------------------
(02) %p_tt/14(+ND, +NT, +NTW, +ER, +EF, +WS, +Abs, +DC, +T, +P, +W, +SS,
(03)                                                        -Timetabling, -EH).
(04) p_tt(ND, NT, NTW, ER, EF, WS, Abs, DC, T, P, W, SS, Timetabling, EH) :-
(05)  set_prop(P, NT, P1, P3),
(06)  team_assign(ND, NT, NTW, ER, WS, Abs, DC, P1, D, E, OAbs),
(07)  tt_split(ND, NT, NTW, WS, OAbs, DC, D, E, TT, TTList, DCList, H),
(08)  zipWith_tt_solve(NTW, WS, EF, T, H, W, SS, TTList, DCList, P3, EHList),
(09)  sum(EHList, #=, EH),
(10)  zipWith_tt_map(NT, NTW, Dk, TT, Timetabling).
(11) ---------------------------------- TOY(FD) -----------------------------------
(12) p_tt:: int -> int -> int -> int -> int -> [[int]] -> [(int,int)] -> [int] ->
(13)                         int -> bool -> labOrder -> labStrat -> ([[int]],int)
(14) p_tt ND NT NTW ER EF WS Abs DC T P W SS = (Timetabling, EH) <==
(15)  P,
(16)  team_assign ND NT NTW ER WS Abs DC == (D, E, OAbs),
(17)  tt_split ND NT NTW WS OAbs DC D E == (TT, TTList, DCList, H),
(18)  sum (zipWith (tt_solve NTW WS EF T H W SS) TTList DClassList) (#=) EH,
(19)  zipWith (tt_map NT NTW) D TT == Timetabling
```

Figure 7.5: CLP($\mathcal{FD}$), CFLP($\mathcal{FD}$): ETP `p_tt` Predicate/Function

riables can be directly used in `tt_split` and `tt_map` (lines 7 and 17, respectively) as, at that point of the computation, the variables will be unified to integer values.

The nature of these systems (allowing them to reason with models) and their use of the $\mathcal{H}$ solver allow them to fit the coordination of the p_tt stages proposed before. A `labeling` primitive placed at the end of `team_assign` (`p_tt_ta_7`) ensures the backtracking to that point when the stages `tt_split`, `tt_solve` and `tt_map` are computed. This backtracking restores all the variables to the point they were in `p_tt_ta_7`. In particular, $Table$ is restored to be a list of lists of fresh logic variables (which are not taken into account by the $\mathcal{FD}$ solver until an $\mathcal{FD}$ constraint involves them). Thus, splitting `Table` into `tt` (in tt_split) simply implies the variable unification between the `Table` and `tt[i]` mate representations of the same logic variable.

Finally, the use of multiple solvers affects the way batch propagation is simulated in CLP($\mathcal{FD}$) systems, breaking the pure declarative view they provide. Now, each solver needs a dedicated flag, as once it is unified to a value (to trigger the frozen constraints just before the solver starts a search) the flag cannot be reused in further solvers. Thus, now `set_prop` (line 5) receives P and NT, generating P1 and P3 (to be a zero value or a fresh logic variable; and a list of NT zero values or a list of NT fresh logic variables, respectively). Whereas P1 is passed to `team_assign` (line 6), P3 is passed to `zipWith_tt_solve` (line 8), assigning each element of P3 to a different working team.

7.3 ETP: Data Structures

The use of dynamic data structures, as vectors in C++ CP($\mathcal{FD}$) systems and lists in CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) ones, eases the formulation of the ETP problem w.r.t. algebraic CP($\mathcal{FD}$) systems, where static array data structures are used. For example, referring back to the instance example ETP-7 of Section 5.1, it contains two kind of days: Working days (with a 20, 22 and a 24 working hours shift) and weekends (with two 24 hours shifts). First, dynamic data structures can be declared empty (to be further filled), but arrays cannot, thus requiring extra input parameter variables to set the dimensions of the array. Second, the unbalanced number of shifts of both kind of days is inherently supported by dynamic data structures, but it is not in arrays. Thus, void shifts of 0 hours are used for filling the empty spaces of those kind of days having less shifts (as in the weekend days, with `[24,24,0]`). This behavior affects to input arguments, but also to computed data structures. For example, when computing $oabs$, it is not known in advance the amount of absences each day contains. Thus, the length of the array is set to an upper bound of `nT*nTW` and, again, void 0 values are used to fill the empty spaces (as in `[1,2,5,6,7,10,11,12,0,0,0,0]` for $oabs_1$).

When dealing with $\mathcal{FD}$ data structures, these additional void values represent additional $\mathcal{FD}$ variables and $\mathcal{FD}$ constraints to be posted to the solver. Referring back again to ETP-7, `p_tt_sp_4` computes $tt = [tt_1, tt_2, tt_3]$ and $dc\_list = [dc\_list_1, dc\_list_2, dc\_list_3]$, where $tt_1 = [[0,0,a,b,c], [l,m,n,o,0], [v,w,x,y,0]]$, $tt_2 = [\ [d,e,f,g,0], [p,q,r,s,0]]$ and $tt_3 = [\ [h,i,j,k,0], [t,0,0,0,u]]$ (respectively $dc\_list_1 = [1,1,2]$, $dc\_list_2 = [1,1]$ and $dc\_list_3 = [1,2]$) are of different length. Thus, in algebraic CP($\mathcal{FD}$) systems a new input parameter variable `dpt = ((nD-1)/nT)+1` is computed, setting the length of all these tt_i and dc_list_i structures. In this concrete example, it forces tt_2 and tt_3 (respectively dc_list_2 and dc_list_3) to contain a new kind of void day they must deal with. Regarding tt_i, this void kind of day is easily represented as a day with $ntw + 1$ shifts of 0 hours to be accomplished (i.e., days where all the variables are directly bound to 0). Regarding dc_list_i, this void kind of day is represented as a new class of day (in the instance as 3, different from working days 1 and weekends 2). Anyway, besides the additional $\mathcal{FD}$ variables and constraints, the modeling of these void days represents a difficulty itself, arisen due to the data structures being used.

On the other hand, the use of arrays provide algebraic CP($\mathcal{FD}$) systems free access to n-dimensional arrays and free indexing by parameter variables. Figure 7.6 presents the access to the bi-dimensional data structure atd (left part), as well as the indexing of the one-dimensional lws with an element of the one-dimensional dc (right part). In both cases, the algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems are presented in order. In the case of C++ systems (line 5), they require $n - 1$ auxiliary elements to access or index an n-dimensional vector. Finally, CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems (lines 7 and 10, respectively) require n auxiliary elements to access or index an n-dimensional vector (but the use of higher order functions in CFLP($\mathcal{FD}$) saves the

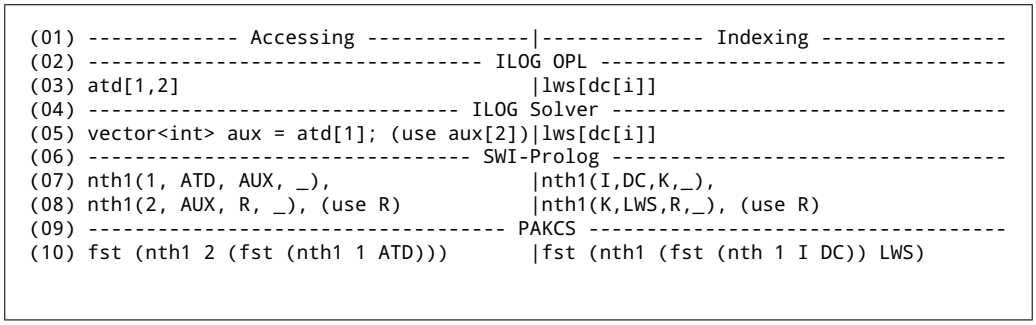

Figure 7.6: Data Structures Accessing and Indexing

explicit declaration of these auxiliary elements). Anyway, they also require using the auxiliary primitive predicate/function `nth1`, which given and index i and a list l returns the *i*-th element of l and the remaining list nl (after removing the *i*-th element from it).

7.4 ETP: Variables

The amount of $\mathcal{FD}$ variables each paradigm needs to formulate the ETP depends on the concrete data structures being used (already presented before), but also on other aspects derived from the constraint solver API and the operational semantics of the system. As an example guiding the section, Figure 7.7 presents the initialization of tt and $trans\_tt$ in MiniZinc, Gecode, SICStus and $\mathcal{TOY}(\mathcal{FD})$ (ILOG OPL, ILOG Solver, SWI-Prolog and PAKCS have been omitted as they are quite similar). These are the main variables of tt_solve, representing the shifts the different workers are assigned to. They are mate variables, with tt ordering the shifts of the team by days $tt \equiv [[w_1(d_1), \ldots, ew(d_1)], \ldots, [w_1(d_{nd/nt}), \ldots, ew(d_{nd/nt})]]$ and $trans\_tt$ ordering the shifts of the team by workers $trans\_tt \equiv [[w_1(d_1), \ldots, w_1(d_{nd/nt})], \ldots, [ew(d_1), \ldots, ew(d_{nd/nt})]]$. The use of both representations eases the modeling of p_tt: Whereas tt are suitable for `p_tt_so_2`, $trans\_tt$ are suitable for `p_tt_so_5`, `p_tt_so_6` and `p_tt_so_7`. The way each paradigm deals with these variables is discussed separately.

7.4.1 Algebraic CP($\mathcal{FD}$)

Two bi-dimensional `arrays` are used to declare `tt` (line 30) and `trans_tt` (line 32). Each representation is independent, in the sense that `tt[i,j]` and `trans_tt[j,i]` use their own memory space and have their own domain and associated constraints. The two mate variables are then linked via an explicit `constraint` (line 34), where the use of `forall` aggregations allow to link all the mate variables in just one line of code.

```
(01) -------------- SICStus --------------|-------------- Gecode -------------------
(02) %gen_transpose/2(+TT, -TransTT).     |class StageIII : public Space {
(03) gen_transpose([[]|_], []).           |protected:
(04) gen_transpose([L|Rl], [T|Rt]) :-     | IntVarArray tt;
(05)     map_headers([L|Rl], T, R),       | IntVarArray trans_tt;
(06)     gen_transpose(R, Rt).            |public:
(07) %                                    | StageIII(bool share, StageIII& s) :
(08) %map_headers/3(+LA, -H, -LB).        |                    Space(share, s) {
(09) map_headers([], [], []).             |  this->tt.update(*this, share, s.tt);
(10) map_headers([L|Rl], [E|Re], [R|Rr]):-|  this->trans_tt.update(*this, share,
(11)     nth1(1, L, E, R),                |                          s.trans_tt);
(12)     map_headers(Rl, Re, Rr).         | }
(13) -------------- TOY(FD) --------------| void init_vars(int nD, int nTW) {
(14) gen_transpose:: [[A]] -> [[A]]       |  //Init tt
(15) gen_transpose = foldr aux_trans []   |  this->tt = IntVarArray(*this,
(16) %                                    |                   nD*(nTW+1), 0, 24);
(17) aux_trans:: [A] -> [[A]] -> [[A]]    |  //Init trans_tt
(18) aux_trans Xs Xss =                   |  IntVarArgs _vars(0);
(19)     zipWith (:) Xs (Xss ++ repeat [])|  for (int i = 0; i < (nTW+1); i++){
(20) %                                    |   for (int j = 0; j < nD; j++){
(21) foldr:: (A->B->B) -> B -> [A] -> B   |    IntVar v(this->tt[j*(nTW+1)+i]);
(22) foldr F Z [] = Z                     |   _vars << v;
(23) foldr F Z [X|Xs] = F X (foldr F Z Xs)|   }
(24) %                                    |  }
(25) zipWith:: (A->B->C) -> [A]->[B]->[C] |  this->trans_tt =
(26) zipWith Z [] Bs = []                 |           IntVarArray(*this, _vars);
(27) zipWith Z [A|As] [] = []             | }
(28) zipWith Z [A|As] [B|Bs] =            |...}
(29)             [Z A B| zipWith Z As Bs] |-------------- MiniZinc -----------------
(30) %                                    |array[1..nD, 1..(nTW+1)]
(31) repeat:: A -> [A]                    |                    of var 0..24: tt;
(32) repeat X = [X|repeat X]              |array[1..(nTW+1), 1..nD]
(33)                                      |              of var 0..24: trans_tt;
(34)                                      |constraint forall (i in 1..(nTW+1), j
(35)                                      | in 1..nD) (trans_tt[i,j] = tt[j,i]);
```

Figure 7.7: Initializing tt and $trans\_tt$ Variables

7.4.2 C++ CP($\mathcal{FD}$)

Two explicit one-dimensional `IntVarArray` are used to declare `tt` and `trans_tt` as attributes of the `StageIII` class (lines 4 and 5). As a difference to the `Golomb` class (cf. Chapter 7.1), where the constructor initialized the variables, posted the constraint network and specified the search strategy, in `StageIII` these tasks are uncoupled from the class constructor and performed in dedicated functions. In concrete, the function `init_vars` (line 13) sets the size and initial domain of `tt` and `trans_tt`, and the copy constructor (line 7) updates these variables to perform each `Space` cloning during search exploration.

Regarding `init_vars`, it initializes two dependent `tt` and `trans_tt` variable data structures (lines 15 and 25, respectively) That is, `tt` and `trans_tt` are declared as different `IntVarArray` objects, but each mate `tt[i]` and `trans_tt[j]` `IntVars` rely on a unique and shared `IntView` variable (the representation being used by the

constraint solver for constraint propagation). To do so, `tt` is created first as a one-dimensional `IntVarArray` of `nD*(nTW+1)` variables. Then, an auxiliary `IntVarArgs _vars` (line 18) argument array (cf. Section 7.1.2) is used to build up `trans_tt` via the variables of `tt`. That is, each `IntVar` of `trans_tt` is just a pointer to the one of `tt`.

Representing `tt` and `trans_tt` as bi-dimensional variable data structures `vector<IntVarArray>` would have eased the formulation of the problem. However, for efficiency reasons, one-dimensional `IntVarArrays` are used (lines 4 and 5), thus implying a harder modeling by recomputing the indexes of the variables involved on each posted constraint. The idea is that the search procedure is the most relevant issue affecting solving efficiency. As the copy constructor is executed on each clone performed during the search exploration, then its execution cost turns crucial as well. Figure 7.8 presents how the copy constructor method would have looked like if a bi-dimensional `tt` had been used. By comparing the approaches of Figure 7.7 and 7.8 it can be seen that, whereas the running time of the copy method based in one-dimensional variables is of $\Theta(1)$, the one of the bi-dimensional variables is of $\Theta(n)$. Obviously, it still has to copy the same number of elements. But copying a single array will be more efficient in hardware, since all the data will be contiguous in memory and parallel bus transfers will be possible. Both Gecode versions are included in the available models, and Chapter 8 proves that the one-dimensional model clearly outperforms the efficiency of the two-dimensional one.

7.4.3 CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$)

In these systems, both `tt` and `trans_tt` are managed as bi-dimensional lists. Whereas `tt` is created by generating $nd * nt$ new fresh logic variables, `trans_tt` is created by transposing `tt` via the predicate (respectively polymorphic function) `gen_transpose` (lines 2 and 14, respectively). They generate each `trans_tt[j,i]` variable by pattern matching on its mate `tt[i,j]`. The semantics of this pattern matching differs depending on the two variables involved: If both are fresh logic variables (or one is an

```
StageIII(bool share, StageIII& s) : Space(share, s) {
 int size = s.tt.size();
 for (int i = 0; i < size; i++){
  IntVarArray aux_tt(s.tt[i]);
  this->tt.push_back(aux_tt);
  this->tt[i].update(*this, share, s.tt[i]);
 }
 ...
}
```

Figure 7.8: Alternative copy Method

$\mathcal{FD}$ variable but the other one is a fresh logic variable), then the matching is managed by the $\mathcal{H}$ solver, which unifies both variables. Otherwise, the matching is managed by the $\mathcal{FD}$ solver, which constrains both $\mathcal{FD}$ variables to be equal by posting an equality constraint to the $\mathcal{FD}$ store. It is important to remark that any fresh logic variable is implicitly attributed as $\mathcal{FD}$ by the system as soon as an $\mathcal{FD}$ constraint involves it.

Thus, as it happened in C++ CP($\mathcal{FD}$) systems, `tt` and `trans_tt` are treated as dependent variable lists. That is, `trans_tt` is built up via `tt`, with each `trans_tt[j,i]` being unified (by the $\mathcal{H}$ solver) with `tt[i,j]`. However, Figure 7.9 presents a fragment of the SICStus predicate gen_d (line 3), implementing p_tt_ta_4, and for which the order in which the clauses are executed affects to the amount of $\mathcal{FD}$ variables created. The predicate creates the list D of ND new fresh logical variables, uses the auxiliary predicate work_each_n_days (line 2) to unify each D[i] with D[i+NT], and finally posts an initial domain over the variables of D (line 6), with a `freeze` predicate controlled by the parameter P of gen_d. By executing the clauses in this order, the unification of D variables remains only NT different logical variables in the list, which then are turned into NT different $\mathcal{FD}$ variables. But, if the domain constraint were posted before the unification, then the ND variables of D would turn into ND different $\mathcal{FD}$ variables, and in this context the pattern matching would post an $\mathcal{FD}$ equality constraint between each pair of $\mathcal{FD}$ variables D[i] and D[i+NT].

Thus, from the point of view of efficiency, it is a good idea to model a problem by unifying first (using $\mathcal{H}$) as many logical variables as possible, before they are turned into $\mathcal{FD}$ variables. However, postponing as much as possible the posting of the $\mathcal{FD}$ constraints when modeling a problem clearly breaks the pure declarative view of modeling CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) paradigms provide.

7.5 ETP: Constraints

The different modeling techniques provided by each paradigm play a role in the posting of the $\mathcal{FD}$ constraints. In this section, the ETP is used to discuss the posting of some constraints related to the stage tt_solve. First, Figure 7.10 presents the step

```
(01) ---------------------------------- SICStus ------------------------------------
(02) %gen_d/4(+ND, +NT, +P, -D).           |%work_each_n_days/2(+N, +L).
(03) gen_d(ND, NT, P, D) :-                |work_each_n_days(N, L) :-
(04)  length(D, ND),                       | nth1(1, L, H1, R1),
(05)  work_each_n_days(NT, D),             | nth1(N, R1, H1, _),
(06)  freeze(P, domain(D, Min, Max)),      | work_each_n_days(N, R1).
(07)  ...                                  |
```

Figure 7.9: Pure Declarative View of Modeling vs. Safe $\mathcal{FD}$ Variables

```
(01) ---------------------------------- MiniZinc ----------------------------------
(02) int: nD;
(03) int: dj;
(04) int: mj;
(05) array[1..dj,1..mj] of 0..24: ws;
(06) array[1..nD] of 1..dj+1: dC;
(07) array[1..nD, 1..(nTW+1)] of 0..1: tt_info;
(08) %
(09) array[1..nD, 1..(nTW+1)] of var 0..24: tt;
(10) array[1..(dj+1)] of set of int: dom_pj =
(11)            [{0} union {ws[i,j] | j in 1..mj where i <= dj} | i in 1..dj+1];
(12) constraint forall (i in 1..nD, j in 1..(nTW+1))
(13)                    (if (tt_info[i,j] = 0) then tt[i,j] = 0 else true endif);
(14) constraint forall (i in 1..nD, j in 1..(nTW+1)) (tt[i,j] in dom_pj[dC[i]]);
(15) ---------------------------------- Gecode ------------------------------------
(16) void post_dom(vector<vector<int>> ws,   |  int s2 = sh.size();
(17)               vector<int> dC, int ntW){|  IntArgs _vals(s2+1);
(18)  for (int i = 0; i < dC.size(); i++){   |  _vals[0] = 0;
(19)   //1. Collect the vars                |  for (int j = 1; j <= s2; j++)
(20)   IntVarArgs _vars(0);                 |   _val[j] = (sh[j-1]);
(21)   for (int z = 0; z < (nTW+1); z++){   |  IntSet vals(_vals);
(22)    IntVar _var(this->tt[i*(nTW+1)+z]); |  //3. Post the domain to vars
(23)    _vars << _var;                      |  for (int j = 0; j < (nTW+1); j++)
(24)   }                                    |    dom(*this, vars[j], vals);
(25)   IntVarArray vars =                   | }
(26)             IntVarArray(*this, _vars); |}
(27)   //2. Collect the domain              |
(28)   std::vector<int> sh = ws[(dC[i]-1)]; |
(29) ---------------------------------- SICStus -----------------------------------
(30) %prep_post_dom/4(+NTW, +WS, +Vars, +DC).|%gen_set/3(Values, +Set, -ResSet).
(31) pre_post_dom(NTW, WS, Vars, DC) :-      |gen_set([], S, S).
(32)  nth1(DC, WS, A0),                      |gen_set([V|Rv], S, NS) :-
(33)  append([0], A0, A1),                   | fdset_parts(S1, V, V, S),
(34)  sort_list(A1, A2),                     | gen_set(Rv, S1, NS).
(35)  reverse(A2, A3),                       |
(36)  empty_fdset(S),                        |%post_domain/2(+Vars, +Values).
(37)  gen_set(A3, S, NS),                    |post_domain(_, [], _).
(38)  post_domain(Vars, NS).                 |post_domain([V|Rv], D) :-
(39)                                         |    V in_set D,
(40)                                         |    post_domain(Rv, D).
(41) ---------------------------------- TOY(FD) -----------------------------------
(42) %post_working_slots:: int -> [[int]] -> [int] -> int -> bool
(43) post_working_slots NTW WS Vars DC = true <==
(44)  domain_valArray Vars (sort_list ([0] ++ (head (drop (DC-1) WS))))
```

Figure 7.10: Posting the Domain of tt Variables (p_tt_so_2)

p_tt_so_2, on which the variables of each day tt_i are initialized with the domain v. Then, Figure 7.11 presents the step p_tt_so_6, on which the variables cv are constrained to be in the domain $-T \ldots T$. To guide the explanation, the third team of the instance ETP-7 (cf. Section 5.1) is referred back. On it, $tt = [[H, I, J, K, 0], [T, 0, 0, 0, U]]$ and $dc = [1, 2]$ (corresponding to the days 3 (Wednesday) and 6 (Saturday) of the timetable). As $ws = [[20, 22, 24], [24, 24]]$ and $ntw = 4$, the domain for $dc_i = 1$ and $dc_i = 2$ must be $[0, 20, 22, 24]$ and $[0, 24]$, respectively. Also, the distribution $(values, cardinalities)$ for $dc_i = 1$ and $dc_i = 2$ must be $[(0, 2), (20, 1), (22, 1), (24, 1)]$ and $[(0, 3), (24, 2)]$, respectively. Finally, the cv variables for each kind of shift (in ETP-7 there are four kind of shifts, of $0, 20, 22, 24$ hours, respectively) and regular worker (in ETP-7, the four workers of team t_3 are w_9, w_{10}, w_{11}, w_{12}) are computed in p_tt_so_5: $cv0_{w9}$, $cv0_{w10}$, $cv0_{w11}$, $cv0_{w12}$ ... $cv24_{w9}$, $cv24_{w10}$, $cv24_{w11}$, $cv24_{w12}$. Given those variables, $T = 1$ is used in p_tt_so_6 to tighten the distribution of shifts: $aux0_1 = cv0_{w9} - cv0_{w10}$, $aux0_2 = cv0_{w9} - cv0_{w11}$, $aux0_3 = cv0_{w9} - cv0_{w12}$, $aux0_4 = cv0_{w10} - cv0_{w11}$, $aux0_5 = cv0_{w10} - cv0_{w12}$ and $aux0_6 = cv0_{w11} - cv0_{w12}$, with the domain of $\{aux0_1, aux0_2, aux0_3, aux0_4, aux0_5$ and $aux0_6\}$ in $-1 \ldots 1$. The same constraints are posted for $aux20_i$, $aux22_i$ and $aux24_i$.

7.5.1 Algebraic CP($\mathcal{FD}$)

Due to the use of void days (cf. Section 7.3), the scheduling of tt_3 in `tt_solve.dzn` uses as input parameters `nD = 3`, `dC = [1,2,3]` and `tt_info = [ [1,1,1,1, 0], [1, 0, 0, 0, 1], [0,0,0,0,0] ]` (instead of the expected $nd = 2$, $dc = [1, 2]$ and $tt = [[H, I, J, K, 0], [T, 0, 0, 0, U]]$ presented before). Once again, the use of void days does not affect the computational effort to schedule the team. Figure 7.10 shows that the first `constraint` block (line 12) concerns with binding to 0 any variable identified as an absence by `tt_info` (which includes any variable of a void day). An `if-then-else` instruction is used, where an idle `true` is used for the `else` part. However, the use of void days affects to the modeling of the domain initialization of the variables. Figure 7.10 shows the computation of the initial domain for `tt`. It relies on the auxiliary `array[1..(dj+1)] of set of int:` `dom_pj` (line 10), where each `dom_pj[i]` represents the set of different kind of shifts a worker can be assigned to in a day of kind `dC[i]`. As it can be seen, each `dom_pj[i]` can be simply obtained as the `union` of 0 (shift for absences or resting) with the different elements of `ws[i,_]`. As there is no ws representation for the void days, it is computed as `{0} union { }` (by using a `where` clause to skip accessing to ws when computing `dom_pj[dj+1]`). Finally, a `forall` aggregation is used in the second `constraint` block (line 14) to elegantly post the domain of all the `tt` variables at a time. On each one, the concrete set of `dom_pj` to be used is selected by indexing it with `dC[i]`. As it can be seen, MiniZinc allows the initialization of a `0..24 var` (as the ones of `tt`) to a set of values by simply using the `in` expression.

```
(01) ---------------------------------- ILOG OPL -----------------------------------
(02) var int tight_vars[1..n_dif_sl, 1..(((nTW-1)*nTW) / 2)] in (-t)..t;
(03) forall (i in 1..n_dif_sl, j in 1..(nTW-1), k in 1..(nTW-j)) (tight_vars[i,
(04)       ((nTW-1)*(j-1) + ((k+j)-1)) - (((j-1)*j) / 2)] = cv[i,j] - cv[i,(j+k)]);
(05) --------- ILOG Solver ---------------------------------- PAKCS----------------
(06) void post_tight(bool prop, int nTW,   |post_tight:: Int -> [Int] -> Bool
(07)      IloEnv& env, IloIntVarArray& cv,|post_tight tight cv | domain (gen_diff
(08)        IloModel&, IloSolver& solver){|                             cv) (-t) t
(09)  bool satisf = true;                 |                       = True
(10)  IloIntVarArray tight_vars(env);     |gen_diff:: [Int] -> [Int]
(11)  for (int j = 0; j < (nTW-1); j++)   |gen_diff [] = []
(12)   for (int k = (j+1); k < nTW; k++)  |gen_diff (x:xs) = (aux_gd x xs)
(13)    if (satisf){                      |                         ++ (gen_diff xs)
(14)     IloIntVar aux(env, (-t), t);     |%
(15)     tight_vars.add(aux);             |aux_gd:: Int -> [Int] -> [Int]
(16)     IloConstraint ct =               |aux_gd _ [] = []
(17)              aux == (cv[j] - cv[k]); |aux_gd x (y:xs) = ((x -# y):
(18)     model.add(ct);                   |                           (aux_gd x xs))
(19)     if (prop)                        |
(20)      satisf = solver.propagate(ct);  |
(21)    }                                 |
(22) }                                    |
(23) ---------------------------------- SWI-Prolog ----------------------------------
(24) %post_tight/3(+P, +T, +CV).          |%map_gen_sub_vars/3(+P, +VL, -NVL).
(25) post_tight(P, T, CV) :-              |map_gen_sub_vars(_, [], []).
(26)  map_gen_sub_vars(P, CV, Tight_Vars), |map_gen_sub_vars(P, [V|Rv], NL) :-
(27)  MT is -T,                           | gen_sub_vars(P, V, Rv, S),
(28)  freeze(P, domain(Tight_Vars, MT, T)).| map_gen_sub_vars(P, Rv, L),
(29)                                      | append(S, L, NL).
(30) %gen_sub_vars/4(+P, +V, +VL, -NVL).  |
(31) gen_sub_vars(_, _, [], []).          |
(32) gen_sub_vars(P, Var, [V|Rv], [S|Rs]):-|
(33)     freeze(P, Var - V #= S),         |
(34)     gen_sub_vars(P, Var, Rv, Rs).    |
```

Figure 7.11: Constraining the Distribution of the Workers (p_tt_so_6)

Orthogonally, Figure 7.11 shows the code of ILOG OPL constraining the cv variables. The explicit bi-dimensional decision variable `array tight_vars` is used (line 2). It ranges in the different kind of shifts `n_dif_sl` and in the amount of pairwise `cv[j]` and `cv[k]` (which are in total $(ntw-1)!$, as it was seen in the instance with $aux0_1 \equiv cv[1] - cv[2]$, ... $aux0_6 \equiv cv[3] - cv[4]$ for the 0 hours kind of shift). The factorial is computed via `((nTW-1)*nTW) / 2`. The variables of `tight_vars` are tightened by simply declaring their initial domain as `(-t)..t` (line 2). Finally, the assignment of the `tight_vars` variables to the subtraction of cv is done via a triple `forall` loop (line 3): Whereas `i` ranges in `n_dif_sl`, `j` and `k` range in `1..(nTW-1)` and `1..(nTW-j)`, to respectively represent the indexes of the pairwise different variables subtracted by `cv[i,j]` and `cv[i,(j+k)]`. However, neither MiniZinc nor ILOG OPL include the notion of a counter in the `forall` instructions. Thus, the link between each `tight_vars[i,counter]` and the indexes `j` and `k` must be set. In this case, the link is `counter = ((nTW-1)*(j-1) + ((k+j)-1)) - (((j-1)*j) / 2)`, which

is far from being straightforward.

7.5.2 C++ CP($\mathcal{FD}$)

In the case of C++ CP($\mathcal{FD}$) systems, due to the use of `vector` data structures no void days are needed (cf. Section 7.3), and then the `dC = [1,2]` received is consistent with the one presented in the initial sketch of the section.

Figure 7.10 presents the code used in Gecode to initialize the variables. A method `post_dom` is used (line 16), which is local to the class `StageIII` (cf. Section 7.2.2), and thus is called within the `tt_solve` stage by the different teams `slv[i]` (cf. Figure 7.4). The method uses the size of dC to create a `for` loop ranging on each of the days the team has to deal with (line 18). For each day, it collects the variables and initializes their domain. Regarding the variable collection, an explicit `IntVarArray vars` is used (line 25), which relies on the auxiliary `IntVarArgs _vars(0)` for its initialization (line 20). Initially empty, a `for` loop uses the operator << to dynamically add the concrete `tt` variables of the day to `_vars`. As it was seen in Section 7.4.2 (with the case of `tt` and `trans_tt` variables) the constructor `IntVar _x(IntVar& y)` is used (line 22), which treats both y and the new _x as two different pointers of a unique solver variable `IntView`. Finally, as Gecode uses an one-dimensional variable array for `tt`, to identify the concrete index of the variable, the offset `i*(nTW+1)` (discriminating the day) plus `z` (discriminating the concrete worker) is used.

Orthogonally, Figure 7.11 shows the code of the method `post_tight` (line 6), which ILOG Solver uses to tighten the `cv` variables. As a difference with Gecode, the constraint solvers used for `team_assign` and each team in `tt_solve` are all represented as `IloSolvers` (cf. Section 7.2.2). Thus, when solving the third team, the solver `slv[2]` must be passed as an argument to any method interacting with it (and so it must be passed its application environment `IloEnv` and its associated constraint store `IloModel`). Moreover, to refer to each `cv[i]` within `post_tight`, the `IntVarArray cv` must be passed as an argument too. Regarding the content of the method, the use of a local `IntVarArray tight_vars` is needed (line 10), as it supports the dynamic addition of variables. Taking advantage of that, a double `for` loop in `j` and `k` is used (lines 11 and 12), similar to the one of ILOG OPL. However, instead of declaring all the variables prior to the loop, in ILOG Solver a new explicit `IloIntVar aux` is created on each loop iteration, which is added to `tight_vars` (line 15). Thus, there is no counter needed to relate the concrete `tight_vars[i]` being used for each concrete `cv[j]` and `cv[k]`, as it is known that on each iteration this variable is going to be aux, which is the last variable being added to `tight_vars`. Finally, to tighten the domain of aux, it is declared with an initial domain with lower and upper bounds `-t` and `t` (line 14), respectively.

7.5.3 CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$)

The use of `list` data structures makes the received arguments consistent with the ones presented in the initial sketch of the section. Both for the domain initialization and for the tightening of the distribution, the CLP($\mathcal{FD}$) approach is discussed first, pointing out then how the CFLP($\mathcal{FD}$) one improves it.

Figure 7.10 presents the code used in SICStus to initialize the variables. Focusing on the concrete instance, the predicate `prep_post_dom` (line 30) receives `WS = [ [20,22,24], [24,24] ]`, `NTW = 4`, the variables of a concrete day and its concrete kind of day (i.e., a first call to `prep_post_dom` will be in charge of `Vars = [H,I,J,K,0]` and `dC = 1` and a second one of `Vars = [T,0,0,0,U]` and `dC = 2`). The list of values of the domain (for example `A2 = [0,20,22,24]` for the first day) is obtained by using the primitive predicates `nth1` (using DC to discriminate the day in `WS`) and append (to add the 0 shift). Then, the predicate `sort_list` (line 34) orders the shifts (with no repetitions) in an increasing order. However, the SICStus API just provides two primitives to post a domain to a variable: The usual `domain([VarList], Lb, Ub)` and `V in_set D`. Thus, the `[0,20,22,24]` domain list must be turned into the `{0,20,22,24}` domain set (another option would have been to post `domain([VarList], 0, 24)` and then add `#\=` constraints for all values in the range `1..23` but for 20 and 22). To construct the set of values the predicates `empty_fdset(S)` (which unifies S with `{}`) and `fdset_parts(NS, V, V, S)` (which, given a set S and a new value V, creates the new set NS including V on its first position) are used (lines 36 and 33, respectively). Thus, `{0,20,22,24}` is constructed by recursively calling to the predicate `gen_set` (line 30) with the reversed values list `[24,22,20,0]`.

In the case of $\mathcal{TOY}(\mathcal{FD})$, the approach is exactly the same. However, the primitive `domain_valArray` (line 44) allows as its arguments the variable list `[H,I,J,K,0]` (respectively `[T,0,0,0,U]`) and the domain value list `[0,20,22,24]` (respectively `[0,24]`), so that it can avoid the set creation. Besides that, the use of higher order functions allows to model the domain initialization in just one line of code.

Orthogonally, Figure 7.11 shows the code of SWI-Prolog to tighten the CV variables. A method `post_tight` is used (line 24), which, besides the propagation mode P and the tight level T receives the CV list corresponding to a concrete kind of shift. As in ILOG OPL and ILOG Solver, `post_tight` constrains `tight_vars` to be in the domain `(-T)..T` (line 28) and to be the subtraction of a concrete pair of `CV[j]` and `CV[k]` (line 33). But, opposite to them, in SWI, the variables of `tight_vars` are created on the fly, as the different pairwise of `CV[j]` and `CV[k]` are traversed. In this setting, the predicate `map_gen_sub_vars` (line 24) would represent the loop `j` and the auxiliary `gen_sub_vars` the k loop.

In the case of PAKCS, the use of higher order functions allows to compute `tight_vars` on the fly as the list argument of the `domain` constraint (line 7). Once again, the auxiliaries functions `gen_diff` (line 11) and `aux_gd` (line 15) would play

the role of the loops j and k, respectively. But, on the latter one, the use of functional notation in (x -# y):(aux_gd x xs) (line 17) provides an implicit declaration of each of the variables of tight_var.

7.6 ETP: A Final Expressiveness Comparison

Once given a detailed description of different modeling issues, this section points out some of the modeling features each paradigm provides. To this end, it focuses on the computation of the concrete data structure $oabs$, as it is easy enough as to explain from scratch the complete code for computing it, and complex enough as to require most of the modeling features of each paradigm. Figures 7.12, 7.13 and 7.14 present the computation of $oabs$ in MiniZinc, Gecode, SICStus and $\mathcal{TOY}(\mathcal{FD})$, respectively (the code for ILOG OPL, ILOG Solver, SWI-Prolog and PAKCS has been skipped, as it is quite similar). To guide the section, the computation of $oabs$ for the concrete ETP-7 instance (cf. Section 5.1) is used. On it, the $abs = [(1,1),\ (2,1),(5,1),\ (6,1),(7,1),\ (10,1),(11,1),\ (12,1),(5,6),\ (6,6),(7,6),\ (10,6),(11,6),\ (12,6)]$ leads to the result $oabs = [\,[1,2,5,6,7,10,11,12],\,[\,],\,[\,],\,[\,],\,[\,],\,[5,6,7,10,11,12],\,[\,]\,]$. Each paradigm is presented separately, with a brief sketch of the code being presented before enumerating the modeling features to be pointed out.

7.6.1 Algebraic CP($\mathcal{FD}$)

Sketch: The bi-dimensional array of int abs (Figure 7.12, line 3) is passed as an input argument (as well as the int num_abs (line 2), indicating the length of the array). Three auxiliary data structures are used: First, the set of int Workers (line 5), containing the identifier of any possible regular worker, as they are the potentially candidates to be absent (in ETP-7, {1,..,12}). Second, the one-dimensional array of set of Workers aux_abs (line 6), filtering the regular workers that are absent each day (in ETP-7, [{1,2,5,6,7,10, 11,12}, {}, .., {}]). Third, the one-dimensional array of int num_abs_day (line 8), computing the size of each aux_abs[i] (in ETP-7, [8,0, .., 0]).

Then, oabs is declared as a bi-dimensional variable array (line 12). As the amount of absences per day is not known in advance, the upper bound nT*nTW is used to set the size of the array, and the void value 0 is used to fill the remaining positions (cf. Section 7.3). Whereas the first constraint block (line 13) uses num_abs_day[i] = k to assign each oabs[i,z] (with z in 1..k) to the z-th element of the set aux_abs[i], the second constraint block (line 15) assigns each oabs[i,z] (with z in k+1..nT*nTW) to 0.

Modeling features:

a. The use of set as a basic data structure, which also includes the use of array of

```
(01) --------------------------------- MiniZinc -----------------------------------
(02) int: num_abs;
(03) array[1..num_abs, 1..2] of 1..(nT*nTW): abs;
(04) %
(05) set of int: Workers = 1..(nT*nTW);
(06) array[1..nD] of set of Workers: aux_abs =
(07)              [{abs[j,1] | j in 1..num_abs where abs[j,2] = i} | i in 1..nD];
(08) array[1..nD] of 0..(nT*nTW): num_abs_day =
(09)               [(sum (j in 1..num_abs where abs[j,2] = i) (1)) | i in 1..nD];
(10)         [(sum (j in 1..(nT*nTW)) (bool2int(j in aux_abs[i]))) | i in 1..nD];
(11) %
(12) array[1..nD, 1..(nT*nTW)] of var 0..(nT*nTW): oabs;
(13) constraint forall (i in 1..nD, j in 1..(nT*nTW)) ((j <= num_abs_day[i]) ->
(14)              (oabs[i,j] = (let {set of Workers: m = aux_abs[i]} in m[j])));
(15) constraint forall (i in 1..nD, j in 1..(nT*nTW)) ((j > num_abs_day[i]) ->
(16)                                                        (oabs[i,j] = 0));
(17) --------------------------------- Gecode -------------------------------------
(18) void c_oabs(int nD, vector<pair<int,int>>& abs, vector<vector<int>>& oabs){
(19)  int size = abs.size();
(20)  for (int i = 0; i < size; i++)
(21)   swap(abs[i].first, abs[i].second);
(22)  sort(abs.begin(), abs.end());
(23) //
(24)  for (int i = 0; i < nD; i++){
(25)   vector<int> aux;
(26)   oabs.push_back(aux);
(27)  }
(28) //
(29)  for (int i = 0; i < size; i++)
(30)   oabs[(abs[i].first)-1].push_back(abs[i].second);
(31) }
```

Figure 7.12: Algebraic and C++ CP($\mathcal{FD}$) Computation of $oabs$

sets, as in aux_abs (line 6).

b. The use of integer range expressions `a..b`, indicating the set `{a,(a+1),(a+2),..,b}`. Then, either all the elements of the set can be considered, as in `i in 1..nD` (line 6), or just the ones holding a concrete condition are filtered, as `j in 1..num_abs where abs[j,2] = i` (line 8).

c. The use of `sum` expressions to count the amount of elements of a data structure holding a condition. For example, `sum (range where cond) (1)` filters first the elements of range holding cond, and it adds 1 to the sum for each of them (Figure 7.3, lines 6, 7 and 8).

d. The direct access to the API of MiniZinc does not provide a method for obtaining the cardinality of a set (computing each `num_abs_day[i]` requires a filtering as the one presented before). However, MiniZinc allows accessing to the *i*-th element of the set `a`, by simply using `a[i]`. It also allows to check if an element `e` belongs to the set `a`, by simply using `(e in a)` (Figure 7.12, line 10). Finally, it allows to coerce this result to an integer by using the primitive `bool2int`.

e. The request of turning any parameter variable v to be displayed in the `output` block

```
(01) ---------------------------------- SICStus ------------------------------------
(02) %c_oabs/3(+ND, +Abs, -OAbs).          |%repeat_N_times/3(+N, +Arg, -List).
(03) c_oabs(N, Abs, OAbs) :-               |repeat_N_times(0, _, []).
(04)  map_swap(Abs, IAbs),                 |repeat_N_times(N, Arg, [Arg|L]) :-
(05)  spl(IAbs, SL),                       | N > 0,
(06)  repeat_N_times(N, [], Ac),           | N1 is (N-1),
(07)  foldl_select_by_day(Ac, SL, OAbs).   | repeat_N_times(N1, Arg, L).
(08)                                       |
(09) %po/3(+P1, +P2, ?Res).                |%foldl_select_by_day/3(+Ac, +AL, -R).
(10) po((X,Y), (Z,T), true) :-             |foldl_select_by_day(Ac, [], Ac).
(11)  ((X > Z) ; (X = Z, Y > T)).          |foldl_select_by_day(Ac, [(A,B)|RAb], R):-
(12) po((X,Y), (Z,T), false) :-            | select_by_day(Ac, (A,B), NAc),
(13)  ((X < Z) ; (X = Z, Y =< T)).         | foldl_select_by_day(NAc, RAb, R).
(14)                                       |
(15) %spl/2(+L1, -L2).                     |%filter_po/4(+P, +L, +R, -LR).
(16) spl([], []).                          |filter_po(_, [], _, []).
(17) spl([(X,Y)|Xs], L) :-                 |filter_po((X,Y),[(Z,T)|Xs],R,[(Z,T)|L]):-
(18)  filter_po((X,Y), Xs, true, S),       | po((X,Y), (Z,T), R),
(19)  spl(S, L1),                          | !,
(20)  filter_po((X,Y), Xs,false, B),       | filter_po((X,Y), Xs, R, L).
(21)  spl(B, L2),                          |filter_po((X,Y), [_| Xs], R, L) :-
(22)  append(L1, [(X,Y)|L2], L).           | filter_po((X,Y), Xs, R, L).
(23)                                       |
(24) %select_by_day/3(+L1, +P, -L2).       |%swap/2(+P1, -P2).
(25) select_by_day(L, (A,B), NL) :-        |swap((A,B), (B,A)).
(26)  nth1(A, L, Elem, Rest),              |
(27)  append(Elem, [B], NE),               |%map_swap/2(+L1, -L2).
(28)  nth1(A, NL, NE, Rest).               |map_swap([], []).
(29)                                       |map_swap([X|Xs], [A|R]) :-
(30)                                       | swap(X, A),
(31)                                       | map_swap(Xs, R).
```

Figure 7.13: CLP($\mathcal{FD}$) Computation of $oabs$

into a decision ($\mathcal{FD}$) one (as oabs, which is displayed by the `team_assign.mzn` file). It also implies, as a collateral effect, converting into a decision variable any other parameter variable v' depending on v (as ATD and ETD, which require accessing to the obtained oabs).

f. The use of implication constraints c_1 -> c_2 (as in `(j > num_abs_day[i]) -> (oabs[i,j] = 0)`). In particular, c_1 must support reification, and c_2 is just posted when c_1 is entailed to `true` (line 13).

g. The use of local variables, as `m` in the first `constraint forall` block, which represents aux_abs[i] in the expression `oabs[i,j] = m[j]` (line 13).

7.6.2 C++ CP($\mathcal{FD}$)

Sketch: The sketch for computing $oabs$ is the same in C++ CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$). Thus, it is presented for the three paradigms. It uses a method, predicate or function c_oabs (Figures 7.12 (line 18), 7.13 (line 2) and 7.14 (line 2), respectively), receiving nD and abs as input parameters, and computing oabs as a result. To do so, the

```
(01) ---------------------------------- TOY(FD) -------------------------------------
(02) c_oabs:: int -> [(int,int)] -> [[int]]
(03) c_oabs N Abs = foldl select_by_day (take N (repeat [])) (spl (map swap Abs))
(04) %
(05) (|>):: (int,int) -> (int,int) -> bool
(06) (X,Y) |> (Z,T) = (X > Z) \/ (X == Z /\ Y > T)
(07) %
(08) spl :: [(int,int)] -> [(int,int)]
(09) spl [] = []
(10) spl [(X,Y)|Xs] = (spl As) ++ [(X,Y)] ++ (spl Bs) <==
(11)                                          bi_filter ((|>) (X,Y)) Xs == (As,Bs)
(12) %
(13) select_by_day:: [[B]] -> (int,B) -> [[B]]
(14) select_by_day L (A,B) = L1 ++ [E++[B]] ++ L2 <== split_at (A-1) L == (L1,[E|L2])
(15) %                                    |%
(16) map:: (A -> B) -> [A] -> [B]         |take:: int -> [A] -> [A]
(17) map F [] = []                        |take 0 _ = []
(18) map F [X|Xs] = [(F X)|(map F Xs)]    |take N [] = []
(19) %                                    |take N [X|Xs] = [X| take (N-1) Xs] <==
(20) (++) :: [A] -> [A] -> [A]            |                                  (N > 0)
(21) [] ++ Ys = Ys                        |foldl:: (A -> B -> A) -> A -> [B] -> A
(22) [X|Xs] ++ Ys = [X|Xs ++ Ys]          |foldl F Z [] = Z
(23) %                                    |foldl F Z [X|Xs] = foldl F (F Z X) Xs
(24) swap:: (A,B) -> (B,A)                |%
(25) swap (X,Y) = (Y,X)                   |(/\):: bool -> bool -> bool
(26) %                                    |false /\ _ = false
(27) bi_filter:: (A -> bool) -> [A] ->    |true /\ X = X
(28)                             ([A],[A])|%
(29) bi_filter _ [] = ([],[])             |(\/):: bool -> bool -> bool
(30) bi_filter F [X|Xs] = if (F X)        |true \/ X = true
(31)                then ([X|As],Bs)      |false \/ X = X
(32)                else (As,[X|Bs]) <==  |%
(33)      bi_filter F Xs == (As,Bs)       |repeat:: A -> [A]
(34) %                                    |repeat X = [X|repeat X]
(35) gen_v_list:: [A]                     |
(36) gen_v_list = [X| gen_v_list]         |
```

Figure 7.14: CFLP($\mathcal{FD}$) Computation of $oabs$

elements of abs are swapped to sw_abs (from (w_i, d_j) to (d_j, w_i)), and then ordered to ord_sw_abs. Then, a bi-dimensional vector or list oabs is created, initially with nD empty elements. Finally, each (di,wj) of ord_sw_abs is traversed, adding wj to the dith element of oabs.

Modeling features:

a. The use of reference parameters & for the arguments abs and oabs (Figure 7.12, line 18). Thus, oabs is firstly declared (empty) in the context of the main program, and then it is passed as an input argument to c_oabs (but the modifications applied on it in the context of the method are reflected when the execution comes back further to the main function).

b. The use of the standard std library. Its API for vector provides (among others) the methods: size (for getting the size of the vector), push_back (for adding a new element to the vector), and sort (which receives two iterators, pointing at the first and

to the past the end of the `vector`); Its API for `pair<A,B>` provides (among others) the methods: `first` and `second` (returning the first and the second elements of the pair, respectively), and `swap` (for swapping the first and second position content, given that the type of the pair is `pair<A,A>`). Thus, the code for computing `oabs` is simple and clear: A `for` loop (line 20), ranging `i` in `1..size` (the size of `abs`) is needed to swap the elements. Then, the methods `abs.begin()` and `abs.end()` compute the two iterators needed to order the vector by textual order. Another `for` loop (line 24), ranging in `1..nD`, allows to add nD new empty `vectors<int>` to `oabs`. Finally, a `for` loop (line 29), ranging `i` in `1..size`, is needed to traverse the pairs `(di,wj)`. On each iteration, it accesses to the `oabs` concrete day by indexing it with `abs[i].first`, modifying its content by adding `abs[i].second`.

7.6.3 CFLP($\mathcal{FD}$)

Sketch: Besides the sketch of Section 7.6.2, the functions `c_oabs`, `spl` (quicksort variation for type `[(int,int)]`), `select_by_day` and the operator `(|>)` are needed, and they are presented in Figure 7.14 together with the primitive auxiliary functions `map`, `bi_filter`, `swap`, `take`, `foldl`, `repeat` (and the operators `(++)`, `/\` and `\/`) they rely on.

Modeling features:

a. The amount of functions needed. Rather than using a single method as in C++ CP($\mathcal{FD}$), the code for CFLP($\mathcal{FD}$) needs to program four different functions (besides the other ten auxiliary ones they rely on).

b. The use of higher order functions. It allows to develop an elegant, simple and neat formulation for each of these functions. As it can be seen, all the functions have one or at most two rules, with all of them being formulated in just one line of code. Focusing on `oabs` (line 2), it uses multiple higher-order applications to implement the already described *sketch*, as `(map swap Abs)` $\equiv a$, `(spl` a`)` $\equiv b$, `(take N (repeat []))` $\equiv c$, and finally `foldl select_by_day` $c\ b$.

c. The use of extra variables on the right hand side of a function rule. In the function `gen_v_list`, it allows to create a new fresh logic variable (Figure 7.14, line 36).

d. The use of lazy evaluation. In CFLP($\mathcal{FD}$) systems, the function arguments are evaluated to the required extent (the call-by-value used in LP vs. the call-by-need used in FP [147]), as in `take N (repeat [])` (line 3). The evaluation computes the requested list by dealing with the potentially infinite list generated by `repeat`.

e. The use of type declarations, easing the development of secure and maintainable programs. It is important to point out that the type declaration is optional, as the system always infer a type for each function, no matter if it is declared or not. The system also supports polymorphic arguments, as the `[[B]]` and `(int, B)` ones of `select_by_day` (line 14).

f. The use of partial application. The type of `bi_filter` is `(A -> bool) -> [A]`

`-> ([A],[A])` (line 27). In the expression `bi_filter ((|>) (X,Y)) Xs`, the partial application of `(|>) (X,Y)` has in fact the type `(A -> bool)` (i.e., it is acting as a pattern not subject of further evaluation until one of the elements of the list `Xs` is applied on it).

g. The use of pattern matching. It allows to discriminate the rule of a function to be applied. For example, in the case of `take` (line 16), the first two rules use 0 and `[]` to discriminate the cases in which there are no more elements to take or more elements in the list, respectively. Thus, the expression `take A B` tries to apply pattern matching between A with 0 or B with `[]`, in order to execute the first or second rule, respectively. If this is not possible, then the third rule is applied (the condition `N > 0` is requested for the consistency of the function, which otherwise might lead to infinite computations in case of being called with an N smaller than 0). As it was seen in Figure 7.9, pattern matching can also imply the posting of $\mathcal{FD}$ equality constraints (if the variable being evaluated is an $\mathcal{FD}$ variable and the pattern is either an integer value or another $\mathcal{FD}$ variable).

7.6.4 CLP($\mathcal{FD}$)

Sketch: Besides the sketch of Section 7.6.2, the predicates `c_oabs`, `spl`, `select_by_day` and swap play the same role as their mate CFLP($\mathcal{FD}$) functions (respectively for the operator `(|>)`). The remaining predicates po, `repeat_N_times`, `foldl_select_by_day`, `filter_pair _ord` and `map_swap` represent the parameterized versions of the CFLP($\mathcal{FD}$) functions `repeat`, `foldl`, `filter` and `map`, respectively.

Modeling features:

a. The amount of predicates needed. As it can be seen, in the computation of $oabs$ there is an equivalence between each CLP($\mathcal{FD}$) predicate and each CFLP($\mathcal{FD}$) function.

b. The less compact definition of each predicate. The lack of higher order functions makes the code of each predicate not as compact as the one of its mate function, as in `c_oabs` (Figure 7.13, line 2), where four lines of code are needed).

c. The amount of variables used. The lack of higher order functions makes explicit the declaration of the variables `IAbs`, SL and Ac (lines 4, 5 and 6, respectively), which were omitted in CFLP($\mathcal{FD}$) (as they were implicitly represented as the underlying result of higher order computations).

d. The amount of predicates used. The lack of polymorphic arguments implies creating a specific predicate for each application of a higher order CFLP($\mathcal{FD}$) function (as `foldl`, `filter` or `map`). For example, the clauses of the predicate `filter_po` (line 15) are dependent on the arguments of the predicate po (line 9). So, for any other filter application in the CFLP($\mathcal{FD}$) model, a new dedicated `filter'` predicate must be explicitly defined.

e. The use of input/output arguments, as the third argument of the predicate po. The

normal use of Res is as output, returning if the pair of the first element is greater than the one of the second. However, in the computation of c_oabs the predicate sort_pair_list uses explicitly true (line 18) and false (line 20) to set the property to be handled by the pairs in the call to filter_po. This true or false value is used then as an input parameter to po, discriminating by pattern matching the rule to be applied.

f. Applying pattern matching also in the result of a computation. In filter_po((X,Y), [(Z,T)|Xs], false, R), by using Res as an input parameter, only the second clause of po (the one making pattern matching with false) is tried.

g. The use of the cut operator (!) (line 19). The predicate filter_po has three clauses: A first one, for the case in which there are no more elements on the list to be filtered. A second one, which checks if (X,Y) and (Z,T) hold the result that R passed as argument. Thus, if po((X,Y),(Z,T), R) fails then the second clause of po fails too. This will trigger the evaluation of the third clause, which recursively computes L as a result by applying filter_po on the rest of elements of the list. Otherwise, if po((X,Y),(Z,T),R) succeeds, then the second clause computes [(Z,T)|L] as a result (where L is again recursively computed).

However, the success of the second clause will make the third one to remain unexplored, and a further backtracking in the goal computation will try to succeed. To avoid this, the operator (!) is used just after the po((X,Y),(Z,T), R) evaluation in the second clause. If it succeeds, then the (!) operator is applied, precluding a further evaluation of the remaining clauses of the predicate (in this case just the third one). In $\mathcal{TOY}(\mathcal{FD})$ there is not such cut operator. Thus, all the rules of a function has to be mutually exclusive, to preclude lazy narrowing evaluation to do pattern matching with some rules. In any case, the lack of a cut operator cannot be seen as a drawback, as it is a non-declarative mechanism (turning relevant the order in which the clauses of a predicate are written).

7.7 Model Sizes

As a final issue for the modeling comparison, this section measures the size of each benchmark for each system. Table 7.1 presents these results, with columns Golomb and Ratio_G representing the lines of code for Golomb, and the ratio of lines of each system w.r.t. the one requiring the minimum number of lines, respectively. Columns ETP and Ratio_E provide the same information, but for ETP. Finally, column Average presents the average values between Ratio_G and Ratio_E, and column Ranking represents the ranking of each system.

There is a clear ranking among the paradigms and systems, and it is stable for both problems. The algebraic CP($\mathcal{FD}$) paradigm is the one requiring the fewest lines of code (although, once again, in ETP the coordination of stages is done manually, thus

System	Golomb	Ratio_G	ETP	Ratio_E	Average	Ranking
MiniZinc	14	1.00	123	1.00	1.00	1
ILOG OPL	17	1.21	138	1.12	1.17	2
$\mathcal{TOY}(\mathcal{FD})$	28	2.00	215	1.79	2.08	3
PAKCS	33	2.36	252	2.05	2.21	4
SICStus	57	4.07	599	4.87	4.47	5
SWI-Prolog	57	4.07	635	5.16	4.62	6
ILOG Solver	104	7.43	762	6.20	6.82	7
Gecode	127	9.07	828	6.73	7.90	8

Table 7.1: Number of Lines per System

not counting the code lines that will be needed for a coordination script). The second one is CFLP($\mathcal{FD}$), requiring 1.8-2.4 more lines of code. The third paradigm is CLP($\mathcal{FD}$), whose code is between 4-5 times greater than the one of the algebraic CP($\mathcal{FD}$). The last paradigm is C++ CP($\mathcal{FD}$), whose code is between 6-9 times greater than the one of the algebraic CP($\mathcal{FD}$). More specifically, when comparing the systems of each paradigm, the differences turn to be much smaller. In algebraic CP($\mathcal{FD}$) systems, ILOG OPL requires between 1.12-1.21 more lines of code than MiniZinc. In CFLP($\mathcal{FD}$) systems, PAKCS requires between 1.15-1.18 more lines of code than $\mathcal{TOY}(\mathcal{FD})$. In CLP($\mathcal{FD}$) systems, SWI-Prolog requires between 1.00-1.06 more lines of code than SICStus. In C++ CP($\mathcal{FD}$) systems, Gecode requires between 1.09-1.22 more lines of code than ILOG Solver.

7.8 Related Work

The literature contains a big amount of documents related with modeling in CP($\mathcal{FD}$). A main introduction to the topic can be found in Chapter 11 of [163], which provides basic notions on how to represent a problem (including variables, constraints and search strategies). It also provides some notions of symmetry breaking, problem reformulation and other techniques to improve the efficiency of the model. Regarding the modeling of CSP's and COP's for the algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems considered in this chapter, perhaps the best reference are the user manuals of MiniZinc [133], ILOG OPL [108], Gecode [173], ILOG Solver [109], SICStus Prolog [126], SWI-Prolog [150], PAKCS [93] and $\mathcal{TOY}(\mathcal{FD})$ [40], which include several case studies with a detailed explanation of the modeling of different problems.

Regarding other modeling comparisons among different CP($\mathcal{FD}$) systems, [71] presents one based on eight C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$) systems. However, the modeling comparison is restricted to a concrete expressiveness aspect, explaining how constraints and meta-constraints can be coded on each of the systems. Similarly, [114] pro-

vides a comparison of 14 constraint solvers, including algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and Mathematical Programming ones. An example of how to encode an `all_different` constraint on each system is provided. However, besides that, the comparison is focused on giving an overview of each solver, considering aspects as the documentation, the amount of examples, and the users community. A benchmark is presented, using different criterion to classify the problems, as the availability of the source code, the number of solutions found and the types of variables used.

Regarding a modeling comparison of CP($\mathcal{FD}$) and other techniques, [99] compares two different ILOG OPL models for an industrial planning problem, using ILOG Solver and ILOG CPLEX [13] as the target solvers, respectively. It concludes that, whereas the CP($\mathcal{FD}$) model leads to a more natural statement of the problem, the computation of bounds for the variable initialization represents a drawback w.r.t. the CPLEX model. Similarly, [53] compares two different ILOG OPL models for an Generic Supply Chain Model, but in this case the ILOG CPLEX ones uses Mixed Integer Programming. It presents the constraints that concrete requirements of the formulation lead to in both models. It shows that the MP model leads to a 20% more variables and constraints than the CP($\mathcal{FD}$) one. Finally, [66] performs a modeling comparison of different problems in both CLP($\mathcal{FD}$) (using SICStus Prolog and B-Prolog [31]) and ASP (using the answer set solvers Smodels [182] and Cmodels [11], the latter relying in the SAT Solver mChaff [138]). It models classic problems, as graph k-coloring, Hamiltonian circuit and generalized Knapsack, and real-life problems as Protein Structure Prediction and a Planning problem. In their analysis, it can be seen that ASP leads to a more compact encoding. In particular, it avoids the explicit use of recursion in most situations, as it happens in the CLP($\mathcal{FD}$) formulation. Moreover, the depth-first search strategy affects the user choices when encoding the algorithm. They also observe that, the more a CLP($\mathcal{FD}$) solution is refined (by introducing further heuristics), the faster is the execution, which does not necessarily happen in the ASP approach.

7.9 Conclusions

Although CFLP($\mathcal{FD}$) is a suitable paradigm for tackling CSP's and COP's, the literature lacks so many practical applications as there are for other well established CP($\mathcal{FD}$) paradigms, as algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$). In this chapter, both the classical CP($\mathcal{FD}$) Golomb Rulers and the real-life ETP COP's have been used to perform an in-depth modeling comparison among the state-of-the-art algebraic CP($\mathcal{FD}$) systems MiniZinc and ILOG OPL, the C++ CP($\mathcal{FD}$) systems Gecode and ILOG Solver, the CLP($\mathcal{FD}$) systems SICStus Prolog and SWI-Prolog, and the CFLP($\mathcal{FD}$) systems PAKCS and $\mathcal{TOY}(\mathcal{FD})$. The conclusions are summarized next.

The constraint solver and its management are transparent to the user in algebraic CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems. But, in the case of the C++ CP($\mathcal{FD}$) systems,

it is needed to develop solver-targeted models, managing the control of its decision variables, constraints, objective function, constraint store, constraint propagation, search engine and search control (as well as the garbage collection of all these elements). More specifically, in Gecode the constraint solver is represented by a `Space` object, and the modeling is done by inheritance, implementing subclasses whose class constructor contains the formulation of the problem. In ILOG Solver, the constraint solver is represented by an `IloSolver` object, but the problem is formulated on the generic modeling layer ILOG Concert, which includes a constraint store and a translation process between the generic variables and constraints to the `IloSolver` targeted ones. Finally, to display the solution, C++ CP($\mathcal{FD}$) systems respectively require specific `Space` and `IloSolver` methods accessing to the values of their attached variables. In the algebraic CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$), as variables are freely declared, they can be straightly used to display their values.

When dealing with a multiple stage formulation, algebraic CP($\mathcal{FD}$) systems may require several models (files). This is the case of ETP, for which algebraic CP($\mathcal{FD}$) systems require one file per each stage, as both $team\_assign$ and tt_solve stages must be isolated. The former because some variables must be treated as decision ones in $team\_assign$ and as parameter ones in tt_split and tt_map. The latter (tt_solve) is isolated to exploit the independence of each team. An external script coordinates the execution of the models, generating the input arguments for each of them. The programming of the script represents a difficulty on its own (besides being a task totally independent from modeling a CP($\mathcal{FD}$) problem). In the rest of paradigms, the model is contained in just one file. In C++ CP($\mathcal{FD}$) systems, the stages tt_split and tt_map are easily implemented by using the C++ abstractions, but the stages $team\_assign$ and each team of tt_solve require a different constraint solver object. This is mandatory, as the C++ CP($\mathcal{FD}$) solvers are not prepared to receive the new constraints of tt_solve when they are already in search mode (previously set in $team\_assign$). Whereas in Gecode the solver abstractions of $team\_assign$ and tt_solve lead to different `Space` subclasses, in ILOG Solver all of them are implemented with `IloSolver` objects. In CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems, the support of model reasoning allows them to better coordinate the different stages of the p_tt formulation. The different stages are coordinated by simply placing them in order, with a $labeling$ primitive at the end of $team\_assign$ and tt_solve ensuring the correct implementation of the architecture of p_tt. Both CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems will internally need to use different solver objects for $team\_assign$ and each team of tt_solve but, as these systems abstract the notion of constraint solver, their management is transparent to the user.

Whereas C++ CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems use dynamic data structures as vectors and lists, algebraic CP($\mathcal{FD}$) systems rely on static arrays. Thus, the latter needs extra input parameters to set in advance the concrete amount of elements each array must contain. Bi-dimensional and three-dimensional arrays are used in tt_split to configure the input parameters and structure of the different teams to be solved

with tt_solve. As these arrays have a fixed length, extra $\mathcal{FD}$ variables and constraints are needed to represent void days in those teams having less days than others. On the other hand, those arrays provide free access and free indexing by using parameter variables, whereas in CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems, additional variables (and even predicate/functions) are needed to access and/or index the elements of the lists.

Saving $\mathcal{FD}$ variables alleviates the constraint network posted to the $\mathcal{FD}$ solver. Taking as an example the mate variables tt and $trans\_tt$ of ETP, it has been seen that algebraic CP($\mathcal{FD}$) systems cannot save any of them, as two independent bi-dimensional arrays must be declared, which are then linked via explicit constraints. In C++ CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems, mate variables can contain a single variable (which is posted to the constraint solver), although maintaining two different modeling variable representations. In C++ CP($\mathcal{FD}$) systems it suffices with initializing each $trans\_tt_{ji}$ using tt_{ij} as its parameter. In particular, in Gecode tt and $trans\_tt$ are represented as one-dimensional variable array attributes of the class `StageIII` (and not as the more naturally bi-dimensional ones used by ILOG Solver). This one-dimensional approach implies a harder modeling, recomputing all the indexes of the variables involved on each of the constraints being posted. However, it allows both tt and $trans\_tt$ to be immediately cloned by the copy constructor, whereas the use of bi-dimensional arrays would have required an additional management, penalizing the performance of the copy method, and thus of the CPU time spent on solving the problem. This represents a low-level issue, showing that the lack of abstraction in C++ CP($\mathcal{FD}$) can result in breaking the isolation between the modeling and solving of a problem. In CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems, tt is built up by generating new fresh logic variables and $trans\_tt$ is constructed from tt by using pattern matching unification. Whereas the unification of two logic variables is done by the $\mathcal{H}$ solver, the equality of two $\mathcal{FD}$ variables posts an $\mathcal{FD}$ constraint to the store. Thus, the unification of tt and $trans\_tt$ must be done before any $\mathcal{FD}$ constraint is posted on them. This clearly breaks the pure declarative view of modeling that CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems claim to provide (as the order in which the problem formulation is declared becomes relevant for the solving efficiency).

The use of aggregations allow algebraic CP($\mathcal{FD}$) systems to declare blocks of constraints in just one line of code, in a much more elegant way than the C++ CP($\mathcal{FD}$) imperative loops and the CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) recursive processes. However, as neither MiniZinc nor ILOG Solver include the notion of a counter for these blocks, the declaration of some constraints turns to be much more difficult (when they imply a complex relation among the indexes of the variables involved). In CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems, by declaring auxiliary variables on the fly, the posting of some constraints has been eased. Moreover, in CFLP($\mathcal{FD}$) the use of higher order functions and functional notation avoid the explicit declaration of such auxiliary variables.

In terms of constraint propagation, algebraic CP($\mathcal{FD}$) just supports batch mode, and the CFLP($\mathcal{FD}$) system PAKCS just incremental mode. C++ CP($\mathcal{FD}$) inherently support batch mode, but their models can be tuned to support incremental by using addi-

tional variables controlling the feasibility of the constraint network after posting each constraint. Similarly, CLP($\mathcal{FD}$) inherently supports incremental mode, but their models can also support batch mode by using additional variables, freezing constraint posting. $\mathcal{TOY}(\mathcal{FD})$ contains two primitives to set the propagation mode, easily supporting the application of different propagation modes to different parts of the model.

Regarding search exploration, a search declaration must specify the variable set, the variable order, the value order and the cost function. Algebraic CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems provide expressive primitives allowing to fully specify them in one line of code. In Gecode, the specification is more low level, attached to the tree exploration being performed, where tree nodes are `Spaces`. A search engine wraps the `Space` to control the search exploration. The copy method of the `Space` must be explicitly programmed (the engine uses it to clone `Spaces` for performing the hybrid recomputation techniques), as well as the cost function (dynamically adding new bound constraints to the tree exploration). Finally, the engine must be forced to look for all the solutions, remaining the last (optimal) one. ILOG Solver requests several primitives to specify the search, to be composed in a final daemon applied (to be executed) to the `IloSolver`. However, neither search control nor specific programming methods are required.

In summary, besides the general interest of comparing those state-of-the-art CP($\mathcal{FD}$) systems, the results have shown $\mathcal{TOY}(\mathcal{FD})$ to be an appealing alternative to any of them, encouraging its use (and the use of the CFLP($\mathcal{FD}$) paradigm itself) for modeling COP's because of a number of advantages:

- It abstracts the notion of the constraint solver, isolating the use of several solvers and the distribution of the constraints on them. It also supports free access to the variables.
- It allows to model the problems in just one file, matching the multiple-stage architecture of the p_tt algorithm by simply placing the stages in order.
- It uses dynamic data structures, and makes handy the access and index of them.
- It allows to save several $\mathcal{FD}$ variables, placing first the variable unifications and then the rest of the $\mathcal{FD}$ constraints.
- It provides batch and incremental primitives for easily applying different propagation modes to different parts of the program.
- It is a declarative general-purpose programming language, including expressive modeling features such as non-deterministic functions, types, higher-order, lazy evaluation, pattern matching and partial application, for allowing the user to write neater formulations. Thus, only the algebraic CP($\mathcal{FD}$) systems require less amount of lines of code for modeling the problems.

Chapter 8

Solving Analysis

Efficiency is a critical issue when choosing a concrete CP($\mathcal{FD}$) system for tackling hard CSP's and COP's. All the expressiveness provided by $\mathcal{TOY}(\mathcal{FD})$ (discussed in Chapter 7) would be meaningless if, when solving different instances of the problem being modeled, the solving performance of $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}s)$ were clearly outperformed by solving native Gecode, ILOG Solver and SICStus `clpfd` models, respectively. This chapter positions the system $\mathcal{TOY}(\mathcal{FD})$ w.r.t. other state-of-the-art CP($\mathcal{FD}$) systems belonging to the Algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$) paradigms. The main purpose of the chapter is to show that $\mathcal{TOY}(\mathcal{FD})$ is competitive w.r.t. any of them for solving different COP's, thus encouraging its use (and the use of the CFLP($\mathcal{FD}$) paradigm itself).

Following Chapter 7, the classical CP($\mathcal{FD}$) problem of Golomb and ETP are used as benchmarks for the algebraic CP($\mathcal{FD}$) systems MiniZinc (using the Gecode solver) and ILOG OPL, the C++ CP($\mathcal{FD}$) systems Gecode and ILOG Solver, the CLP($\mathcal{FD}$) systems SICStus and SWI, and the CFLP($\mathcal{FD}$) systems PAKCS, $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}s)$. The Golomb and ETP instances used in Sections 3.4 and 5.4 are revisited now: G-9 & ETP-7 (whose solving times are of the order of magnitude of tenths of second), G-10 & ETP-15 (seconds) and G-11 & ETP-21 (minutes).

When performing a solving comparison among different systems and problems, many performance metrics can be considered: The execution or CPU time, the memory requirements, the compilation time, etc. The solving comparison of this chapter has two clear targets: Measuring the CPU time of each system for running the benchmark, and provide a low-level monitoring of each $\mathcal{TOY}(\mathcal{FD})$ version and its native constraint solver executions.

The chapter is organized as follows: Section 8.1 sets the context for the solving comparison. Section 8.2 presents the performance of all the systems, analyzing the ranking and slow-down results. It also discusses the performance ranking among the Gecode, ILOG Solver, SICStus `clpfd` and SWI-Prolog `clpfd` constraint solvers, gathering the related systems into different sets. Section 8.3 presents a dedicated ranking and

slow-down analysis of the Gecode related systems MiniZinc, Gecode and $\mathcal{TOY}(\mathcal{FD}g)$. Focusing on G-11 and ETP-21, it discusses the search statistics of the different systems, and provides a low-level monitoring of the search exploration in both $\mathcal{TOY}(\mathcal{FD}g)$ and Gecode, discussing their differences. Sections 8.4 and 8.5 are similar to Section 8.3, but they focus on the ILOG Solver and SICStus `clpfd` related systems, respectively Section 8.6 presents some related work. Finally, Section 8.7 reports conclusions.

8.1 Setting for the Experiments

All the benchmarks are run in a machine with an Intel Dual Core 2.4Ghz processor and 4GB RAM memory. The OS used is Windows 7 Professional SP1 (32 bits.)

The system versions used are: MiniZinc 1.5 (connected to Gecode 3.7.3), ILOG OPL 3.7 (connected to ILOG Solver 6.0), Gecode 3.7.3, ILOG Concert 12.2 & ILOG Solver 6.8, SICStus 3.12.8, SWI-Prolog 6.2.6, PAKCS 1.9.2 (connected to SICStus `clpfd` 3.12.8), $\mathcal{TOY}(\mathcal{FD}g)$ (connected to Gecode 3.7.3), $\mathcal{TOY}(\mathcal{FD}i)$ (connected to ILOG Concert 12.2 & ILOG Solver 6.8) and $\mathcal{TOY}(\mathcal{FD}s)$ (connected to SICStus `clpfd` 3.12.8). Microsoft Visual Studio 2008 tools are used to compile and link the C++ code of the Gecode and ILOG Solver models, as well as the $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ systems.

Besides that, a common framework is set for performing the experiments, considering the global constraints being used, the search strategy and propagation mode, and the measurement of the elapsed time in seconds.

Global Constraints. Golomb and ETP include `all_different`, `count` and `distribute` global constraints. To set a common configuration, the local consistency of them have been set to value consistency (i.e., the constraint only propagates when the domain of one of its associates variables becomes a singleton). Table 8.1 presents the availability and value consistency support per global constraint and system. Each cell in the table displays the pair $i - j$, with i describing if the constraint is available (and, if not, how is it implemented), and j if it supports value consistency. As it can be seen, it is not possible to set a common configuration for all the systems being compared. For example, for `all_different`, whereas ILOG OPL, Gecode, ILOG Solver, SICStus, $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}s)$ support value consistency, MiniZinc, SWI-Prolog and PAKCS do not support it.

Search Strategy and Propagation Mode. In Golomb, its m variables are labeled in textual order, minimizing m_{n-1}. In ETP, for each team tt_i, its $trans\_tt$ variables are also labeled in textual order, minimizing eh. Regarding propagation mode, it is not possible to set up a common framework for all the systems, as PAKCS only supports incremental, and MiniZinc and ILOG OPL only supports batch. Thus, both for Golomb and ETP, incremental mode is selected (except for MiniZinc and ILOG OPL).

Measuring the Elapsed Time. The measurement of the MiniZinc model is done via the Gecode/FlatZinc front-end `fzn-gecode`. It first compiles the MiniZinc model to

System	All-Different	Count	Distribute
MiniZinc	Yes - No	Yes - No	Yes - No
ILOG OPL	Yes - Yes	No (sum) - No	Yes - No
Gecode	Yes - Yes	Yes - Yes	Yes - Yes
ILOG Solver	Yes - Yes	No (sum) - Yes	Yes - Yes
SICStus	Yes - Yes	Yes - No	Yes (global cardinality) - No
SWI-Prolog	Yes - No	No (exactly) - No	No (set of exactly) - No
PAKCS	Yes - No	Yes - No	No (set of count) - No
$\mathcal{TOY}(\mathcal{FD}g)$	Yes - Yes	Yes - Yes	Yes - Yes
$\mathcal{TOY}(\mathcal{FD}i)$	Yes - Yes	Yes - Yes	Yes - Yes
$\mathcal{TOY}(\mathcal{FD}s)$	Yes - Yes	Yes - No	Yes - No

Table 8.1: Constraints Availability and Value Consistency

a FlatZinc one (with a customized compilation of the global constraints). Then, a parser provided by Gecode reads the FlatZinc model, automatically generating an equivalent Gecode `Space`, which is executed. The front-end includes commands for computing one or all solutions, as well as for displaying some statistics (including the CPU time). The measurement of the ILOG OPL model is done within the graphical interface of ILOG OPL Studio. It automatically compiles the model to the suitable ILOG Solver input, executing it and displaying the solutions and some statistics (including the CPU time).

The rest of the systems execute the instances within the SICStus framework, making use of the SICStus predicate `statistics(runtime, [X,_])` [126]. It unifies X to the CPU time used, excluding memory management and system calls (this time also includes the CPU time used for handling the $\mathcal{FD}$ constraint solver). Whereas for SICStus and $\mathcal{TOY}(\mathcal{FD})$ this setup comes for free (as $\mathcal{TOY}(\mathcal{FD})$ is implemented in SICStus Prolog), the Gecode and ILOG Solver models are executed within SICStus via wrapping its C++ main function into a SICStus predicate p (see the SICStus interface to C++ [126]), and then executing p on the SICStus engine. Finally, the SWI-Prolog model also makes use of the `statistics(runtime, [X,_])` predicate, but it is obviously executed within a SWI-Prolog session.

Each instance has been executed five times on each system (to avoid any particular side effect associated to the OS affecting the achieved time on a single run). Then, the best and worse times have been discarded, computing as a final result (to be displayed in the tables) the mean of the remaining three.

Instance Classification. By referring back to tables 3.1 and 5.2, the instances can be classified in two groups: On the one hand, G-9, G-10, G-11 and ETP-21 are classified as *just search: js* instances. For them, the three $\mathcal{TOY}(\mathcal{FD})$ versions spend searching above the 98% of the total CPU time, turning the performance comparison into purely CP($\mathcal{FD}$) dependent. That is, the CPU time of each system directly comes from the

performance of its solver in achieving the pure CP($\mathcal{FD}$) mechanism of performing a search exploration, by propagating basic and global constraints. On the other hand, ETP-7 and ETP-15 are classified as *different factors: df* instances. For them, the three $\mathcal{TOY}(\mathcal{FD})$ versions spend searching a percentage of the total CPU time ranging in a 0%-98%, turning the performance comparison into both CP($\mathcal{FD}$) and paradigm inherent overheads dependent.

Generated Models. Due to low-level details, the built models of the different systems relying on a same constraint solver library will slightly differ. A deeper discussion will be given on this in Sections 8.3 (for MiniZinc, Gecode and $\mathcal{TOY}(\mathcal{FD}g)$), 8.4 (for ILOG OPL, ILOG Solver and $\mathcal{TOY}(\mathcal{FD}i)$) and 8.5 (for SICStus, PAKCS and $\mathcal{TOY}(\mathcal{FD}s)$).

8.2 General Performance Comparison

This section provides a performance comparison of the 10 systems being considered, analyzing their ranking and slow-down results. It also discusses the performance ranking among the Gecode, ILOG Solver, SICStus `clpfd`, and SWI-Prolog `clpfd` constraint solvers, gathering the systems by their underlying solver.

Tables 8.2 and 8.3 present the performance for the Golomb and ETP problems, respectively. Columns `Instance` and `System` represent the concrete instance and system being considered, respectively. The systems are sorted by an increasing CPU time, with column `Ranking` representing the ranking. Finally, columns `Time` and `Sl-Dw` represent the CPU time of each system and its slow-down w.r.t. the fastest one, respectively. The CPU times are measured in seconds, with precision set to two decimal digits (except for the instances G-9 and ETP-7, in which it is set to three digits). The results for G-11 and ETP-21 are not shown for SWI, as the computation did not finish after half an hour.

The general performance results of the 10 systems reveals that, for *js* instances there is a clear performance order among the constraint solvers, with the ranking performance positions 1-3, 4-6, 7-9 and 10 got by the Gecode, ILOG Solver, SICStus related systems and SWI, respectively. For *df* instances, this performance order among the solvers is partially broken. On the one hand, SICStus and ILOG Solver obtain a better ranking for ETP-7 and ETP-15, and PAKCS, $\mathcal{TOY}(\mathcal{FD}s)$ and SWI-Prolog do it as well, but just for ETP-7. On the other hand, both $\mathcal{TOY}(\mathcal{FD}g)$ and ILOG OPL obtain a worse ranking for ETP-7, and both MiniZinc and $\mathcal{TOY}(\mathcal{FD}i)$ obtain a worse ranking for both ETP-7 and ETP-15.

8.2.1 Golomb Slow-down Analysis

First, the slow-down of each system w.r.t. the one with best performance is presented. Then, the three systems using the Gecode constraint solver library are sorted by their

Instance	System	Time	Sl-Dw	Ranking
G-9 *(js)*	MiniZinc	0.109	1.00	1
	Gecode	0.234	2.15	2
	$\mathcal{TOY}(\mathcal{FD}g)$	0.250	2.29	3
	$\mathcal{TOY}(\mathcal{FD}i)$	0.421	3.86	4
	ILOG Solver	0.468	4.29	5
	ILOG OPL	0.500	4.59	6
	SICStus	0.764	7.01	7
	PAKCS	0.810	7.43	8
	$\mathcal{TOY}(\mathcal{FD}s)$	0.842	7.72	9
	SWI-Prolog	29.170	26.76	10
G-10 *(js)*	MiniZinc	0.81	1.00	1
	Gecode	2.11	2.60	2
	$\mathcal{TOY}(\mathcal{FD}g)$	2.11	2.60	3
	$\mathcal{TOY}(\mathcal{FD}i)$	3.56	4.40	4
	ILOG Solver	3.85	4.75	5
	ILOG OPL	4.40	5.43	6
	SICStus	6.65	8.20	7
	PAKCS	7.12	8.79	8
	$\mathcal{TOY}(\mathcal{FD}s)$	7.35	9.07	9
	SWI-Prolog	257.17	317.49	10
G-11 *(js)*	MiniZinc	16.02	1.00	1
	Gecode	41.57	2.59	2
	$\mathcal{TOY}(\mathcal{FD}g)$	42.01	2.62	3
	$\mathcal{TOY}(\mathcal{FD}i)$	72.65	4.53	4
	ILOG Solver	77.08	4.81	5
	ILOG OPL	88.28	5.51	6
	SICStus	143.60	8.96	7
	PAKCS	151.60	9.46	8
	$\mathcal{TOY}(\mathcal{FD}s)$	153.05	9.55	9
	SWI-Prolog	-	-	10

Table 8.2: Golomb Comparison Results

Instance	System	Time	Sl-Dw	Ranking
ETP-7 *(df)*	Gecode	0.030	1.00	1
	SICStus	0.094	3.13	2
	ILOG Solver	0.156	5.20	3
	$\mathcal{TOY}(\mathcal{FD}g)$	0.192	6.40	4
	PAKCS	0.220	7.33	5
	$\mathcal{TOY}(\mathcal{FD}s)$	0.248	8.27	6
	ILOG OPL	0.360	12.00	7
	MiniZinc	0.420	14.00	8
	SWI-Prolog	0.998	33.27	9
	$\mathcal{TOY}(\mathcal{FD}i)$	1.380	46.00	10
ETP-15 *(df)*	Gecode	0.56	1.00	1
	$\mathcal{TOY}(\mathcal{FD}g)$	0.90	1.61	2
	ILOG Solver	0.96	1.71	3
	ILOG OPL	1.00	1.78	4
	MiniZinc	1.30	2.32	5
	SICStus	1.88	3.36	6
	PAKCS	2.36	4.21	7
	$\mathcal{TOY}(\mathcal{FD}s)$	3.22	5.75	8
	$\mathcal{TOY}(\mathcal{FD}i)$	3.28	5.86	9
	SWI-Prolog	43.46	77.61	10
ETP-21 *(js)*	Gecode	49.52	1.00	1
	$\mathcal{TOY}(\mathcal{FD}g)$	50.04	1.01	2
	MiniZinc	75.80	1.53	3
	ILOG Solver	93.42	1.89	4
	$\mathcal{TOY}(\mathcal{FD}i)$	109.02	2.20	5
	ILOG OPL	124.70	2.52	6
	SICStus	190.48	3.85	7
	PAKCS	199.30	4.02	8
	$\mathcal{TOY}(\mathcal{FD}s)$	338.16	6.83	9
	SWI-Prolog	-	-	10

Table 8.3: ETP Comparison Results

ranking, and the three using the ILOG Solver and SICStus `clpfd` constraint libraries are respectively sorted as well. This leads to three sets of three teams. For each instance, a comparison among the best system of each set is done, and the second best systems and worse ones are respectively compared as well.

Slow-down by systems: Table 8.2 shows that the ranking is stable for the three instances, with MiniZinc ranking 1 in all of them. At first glance, it can be surprising to see MiniZinc and $\mathcal{TOY}(\mathcal{FD}i)$ with a better performance than Gecode and ILOG Solver (respectively). Sections 8.3 and 8.4 will provide a deeper discussion on these cases, showing that the different propagation levels of global constraints (for Gecode related systems) and the different models being built-up (for ILOG Solver ones) justify this behavior. Besides that, some conclusions are obtained now from Table 8.2.

The head-to-head slow-down between any system w.r.t. MiniZinc scales as the instances scale up (e.g., the 7.01, 8.20 and 8.96 slow-down of SICStus w.r.t. MiniZinc for G-9, G-10 and G-11, resp). As a direct consequence of this, the slow-down interval achieved, i.e., the interval from the system ranked 2 w.r.t. MiniZinc and the system ranked 9 w.r.t. MiniZinc (SWI-Prolog is left out from the analysis as there is no measure for ETP-21) also scales as the instances scale up: From the 2.15-7.72 for G-9, to the 2.60-9.07 for G-10 and the 2.59-9.55 for G-11.

Regarding the performance ranking among the constraint solvers, the slow-down of the remaining Gecode related systems (w.r.t. MiniZinc) range in 2.15-2.29 for G-9, 2.60 for G-10 and 2.59-2.62 for G-11. The slow-down of the ILOG Solver related systems (w.r.t. MiniZinc) range in 3.86-4.59 for G-9, 4.40-5.43 for G-10 and 4.53-5.51 for G-11. The slow-down of the SICStus related systems (w.r.t. MiniZinc) range in 7.01-7.72 for G-9, 8.20-9.07 for G-10 and 8.96-9.55 for G-11.

Slow-down by constraint solver library: Table 8.4 presents the slow-downs sorted by the constraint solver related systems. Whereas column `Instance` represents the concrete instance, columns `Rank` and `Systems` represent the ranking of the systems and their names, respectively. For example, coming back to Table 8.2, it can be seen that, within the Gecode related systems, their internal ranking is: (1) MiniZinc, (2) Gecode and (3) $\mathcal{TOY}(\mathcal{FD}g)$ (as they are ranked 1, 2 and 3, respectively). Similarly, within the ILOG Solver related systems their internal ranking is: (1) $\mathcal{TOY}(\mathcal{FD}i)$, (2) ILOG Solver and (3) ILOG OPL (as they are ranked 4, 5 and 6, respectively), and within the SICStus `clpfd` related systems their internal ranking is: (1) SICStus, (2) PAKCS and (3) $\mathcal{TOY}(\mathcal{FD}s)$ (as they are ranked 7, 8 and 9, respectively). Thus, the the first row of Table 8.4 represents to the Gecode, ILOG Solver and SICStus systems ranked (1), for the instance G-9. Similarly, the second and third row represents to the systems ranked (2) and (3), respectively, and again for the instance G-9. Finally, columns `Gecode`, `ILOG` and `SICStus` represent the slow-downs of the concrete system w.r.t. to the Gecode related one. These results reveal the following conclusions.

First, the ranking for Golomb is stable in the 3 instances. Thus, regarding the ILOG Solver related systems, Table 8.4 compares $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. MiniZinc (ranking 1), ILOG

Instance	Rank	Systems	Gecode	ILOG	SICStus
G-9	1	MiniZinc, $\mathcal{TOY}(\mathcal{FD}i)$, SICStus	1.00	3.86	7.01
	2	Gecode, ILOG Solver, PAKCS	1.00	2.00	3.46
	3	$\mathcal{TOY}(\mathcal{FD}g)$, ILOG OPL, $\mathcal{TOY}(\mathcal{FD}s)$	1.00	2.00	3.37
G-10	1	MiniZinc, $\mathcal{TOY}(\mathcal{FD}i)$, SICStus	1.00	4.40	8.20
	2	Gecode, ILOG Solver, PAKCS	1.00	1.82	3.42
	3	$\mathcal{TOY}(\mathcal{FD}g)$, ILOG OPL, $\mathcal{TOY}(\mathcal{FD}s)$	1.00	2.09	3.48
G-11	1	MiniZinc, $\mathcal{TOY}(\mathcal{FD}i)$, SICStus	1.00	4.53	8.96
	2	Gecode, ILOG Solver, PAKCS	1.00	1.85	3.65
	3	$\mathcal{TOY}(\mathcal{FD}g)$, ILOG OPL, $\mathcal{TOY}(\mathcal{FD}s)$	1.00	2.10	3.64

Table 8.4: Constraint Solver Library Slow-Down Results for Golomb

Solver w.r.t. Gecode (ranking 2) and ILOG OPL w.r.t. $\mathcal{TOY}(\mathcal{FD}g)$ (ranking 3). Regarding the SICStus related systems, Table 8.4 compares SICStus w.r.t. MiniZinc, PAKCS w.r.t. Gecode and $\mathcal{TOY}(\mathcal{FD}s)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}g)$.

In this setting, the slow-down interval obtained before for G-9, when comparing the ILOG Solver related systems w.r.t. MiniZinc (cf. Table 8.2), was 3.86-4.59. Now, with the new comparison of Table 8.4, column `ILOG` shows that it is reduced to 2.00-3.86, as both the slow-down of ILOG Solver w.r.t. Gecode and the one of ILOG OPL w.r.t. $\mathcal{TOY}(\mathcal{FD}g)$ are 2.00. In the case of SICStus (column `SICStus`), the slow-down interval 7.01-7.72 of Table 8.2 is now reduced to 3.37-7.01, as the slow-down of PAKCS w.r.t. Gecode is 3.46 and the one of $\mathcal{TOY}(\mathcal{FD}s)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}g)$ is 3.37. Thus, it can be seen that, whereas there is a performance order among all the Gecode, ILOG Solver and SICStus related systems, the slow-down for systems ranking 2 and 3 are around a 50% smaller than the ones of the systems ranking 1. This situation also happens for G-10 and G-11, where the slow-down for systems ranking 2 and 3 are between a 50%-60% smaller than the ones of the systems ranking 1.

Second, whereas the slow-down of each system w.r.t. MiniZinc increases as the instances scale up, it does not happen now with the ILOG Solver and SICStus systems ranked 2 and 3 in Table 8.4. For example, the slow-down of ILOG Solver (respectively PAKCS) w.r.t. Gecode decreases from G-9 to G-10, and then it increases from G-10 to G-11.

In summary, it can be seen that Golomb is a relatively homogeneous problem:

- The ranking of any system for any problem instance is fixed.
- The slow-down of any system (w.r.t. the optimal MiniZinc) scales as the instances scale up.
- This slow-down w.r.t. MiniZinc is around 2.0-2.5, 4.0-5.5 and 7.0-9.5 for the Gecode, ILOG Solver and SICStus related systems, respectively.
- However, when the homogeneous ranked systems are compared, the slow-down of ILOG Solver systems w.r.t. Gecode ones decreases to around 2.0-4.5, and the

slow-down of SICStus systems w.r.t. Gecode ones decreases to around 3.3-9.0.

- Moreover, the slow-down of any ILOG Solver (respectively SICStus) related system w.r.t. any Gecode system does not necessarily scale as the instances scale up.

8.2.2 ETP Slow-down Analysis

The same two analysis by systems and constraint solvers are performed for the ETP.

Slow-down by systems: The ETP ranking is not stable for the three instances, but the C++ CP($\mathcal{FD}$) Gecode ranks 1 for all of them. Thus, two comparisons are needed: One by ranking (i.e., Gecode vs. the system ranking 2, 3, etc.) and a head-to-head comparison (i.e., Gecode vs. $\mathcal{TOY}(\mathcal{FD}g)$, ILOG Solver, SICStus, etc.)

Regarding the ranking comparison, the slow-down between the systems w.r.t. Gecode decreases as the instances scale up. For example, the comparison of the systems ranked 2 w.r.t. Gecode, which respectively evolve from the SICStus 3.13 for ETP-7, to the $\mathcal{TOY}(\mathcal{FD}g)$ 1.61 for ETP-15 and the $\mathcal{TOY}(\mathcal{FD}g)$ 1.01 for ETP-21. Moreover, the slow-down interval (considering all the systems) decreases as the instances scale up (both in their lower and upper bound values, and in the difference of upper and lower). It is 3.13-46.00 for ETP-7, then 1.61-5.86 for ETP-15 and, finally, 1.01-6.83 for ETP-21.

Regarding the head-to-head slow-down between each system w.r.t. Gecode, it does not follow a clear pattern as the instances scale up. There are systems for which the slow-down decreases (as for MiniZinc, $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ and PAKCS), increases (as for SICStus and SWI) and decreases from ETP-7 to ETP-15, and then increases from ETP-15 to ETP-21 (as for ILOG OPL, ILOG Solver and $\mathcal{TOY}(\mathcal{FD}s)$). As it can be seen, this does not follow a pattern neither by paradigms nor by constraint solving libraries.

Finally, regarding the performance order of constraint solvers, it can be seen that the slow-down of the remaining Gecode related systems (w.r.t. Gecode) ranges in 6.40-14.00 for ETP-7, 1.61-2.32 for ETP-15 and 1.01-1.53 for ETP-21. The slow-down of the ILOG Solver related systems (w.r.t. Gecode) ranges in 5.20-46.00 for ETP-7, 1.71-5.86 for ETP-15 and 1.89-2.52 for ETP-21. The slow-down of the SICStus related systems (w.r.t. Gecode) ranges in 3.13-8.27 for ETP-7, 3.36-5.75 for ETP-15 and 3.85-6.83 for ETP-21.

Slow-down by constraint solver library: Table 8.5 presents the slow-down sorted by constraint solver related systems (similar to Table 8.4, but for ETP). The general ranking of ETP is not stable for the 3 instances (cf. Table 8.3), but it is stable in the comparison performed in Table 8.5. Regarding the ILOG Solver related systems, it compares ILOG Solver w.r.t. Gecode (ranking 1), ILOG OPL w.r.t. $\mathcal{TOY}(\mathcal{FD}g)$ (ranking 2) and $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. MiniZinc (ranking 3). There is an exception in ETP-21, where ILOG OPL and $\mathcal{TOY}(\mathcal{FD}i)$ switch their respective 2 and 3 rankings. Regarding the SICStus related systems, Table 8.5 compares SICStus w.r.t. Gecode, PAKCS w.r.t. $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}s)$ w.r.t. MiniZinc.

Instance	Rank	Systems	Gecode	ILOG	SICStus
ETP-7	1	Gecode, ILOG Solver, SICStus	1.00	5.20	3.13
	2	$\mathcal{TOY}(\mathcal{FD}g)$, ILOG OPL, PAKCS	1.00	1.88	1.15
	3	MiniZinc, $\mathcal{TOY}(\mathcal{FD}i)$, $\mathcal{TOY}(\mathcal{FD}s)$	1.00	3.29	0.59
ETP-15	1	Gecode, ILOG Solver, SICStus	1.00	1.71	3.36
	2	$\mathcal{TOY}(\mathcal{FD}g)$, ILOG OPL, PAKCS	1.00	1.11	2.62
	3	MiniZinc, $\mathcal{TOY}(\mathcal{FD}i)$, $\mathcal{TOY}(\mathcal{FD}s)$	1.00	2.52	2.48
ETP-21	1	Gecode, ILOG Solver, SICStus	1.00	1.89	3.85
	2	$\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$, PAKCS	1.00	2.18	3.98
	3	MiniZinc, ILOG OPL, $\mathcal{TOY}(\mathcal{FD}s)$	1.00	1.65	4.46

Table 8.5: Constraint Solver Library Slow-Down Results for ETP

First, in ETP-7, it can be seen that the performance ranking among constraint solvers is not (1) Gecode, (2) ILOG Solver and (3) SICStus, but (1) Gecode, (2) SICStus and (3) ILOG Solver, with the three SICStus related systems behaving better than the ILOG Solver ones. Furthermore, in the case of systems ranking 3, the performance order is (1) SICStus, (2) Gecode and (3) ILOG Solver, with $\mathcal{TOY}(\mathcal{FD}s)$ behaving even better than MiniZinc.

In this setting, in ETP-7, a 5.20-46.00 slow-down interval was obtained when comparing the ILOG Solver related systems w.r.t. the C++ CP($\mathcal{FD}$) Gecode (cf. Table 8.3). Now, with the new comparison of Table 8.5, this slow-down interval is reduced to 1.88-5.20, as the slow-down of ILOG OPL w.r.t. $\mathcal{TOY}(\mathcal{FD}g)$ is 1.88 (60% smaller than 5.20) and the one of $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. MiniZinc is 3.29 (40% smaller than 5.20). In the case of SICStus, the slow-down interval 3.13-8.27 of Table 8.3 is now reduced to 0.59-3.13, as the slow-down of PAKCS w.r.t. $\mathcal{TOY}(\mathcal{FD}g)$ is 1.15 (60% smaller than 3.13) and the one of $\mathcal{TOY}(\mathcal{FD}s)$ w.r.t. MiniZinc is 0.59 (80% smaller than 3.13).

Second, in ETP-15 and ETP-21, the performance ranking among constraint solvers becomes (1) Gecode, (2) ILOG Solver and (3) SICStus, and it stands for all the systems ranking 1, 2 and 3. However, the slow-down of this performance order varies for each instance and constraint solver systems related being considered.

In ETP-15 and ILOG Solver related systems, it is observed that the slow-down of the systems ranking 2 (ILOG OPL w.r.t. $\mathcal{TOY}(\mathcal{FD}g)$) is a 35% smaller than the one of those ranking 1 (ILOG Solver w.r.t. Gecode). On the other hand, the slow-down of systems ranking 3 ($\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. MiniZinc) is a 50% greater than the one of those ranking 1. However, ETP-21 the behavior is the opposite. In this case, systems ranking 2 ($\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}g)$) have a 15% greater slow-down than the ones ranking 1 (ILOG Solver w.r.t. Gecode), and systems ranking 3 (ILOG OPL w.r.t. MiniZinc) have a 15% smaller slow-down than the ones ranking 1.

This lack of stable slow-downs might be due to the switch of ILOG OPL and $\mathcal{TOY}(\mathcal{FD}i)$ in ranks 2 and 3 of ETP-15 and ETP-21. However, this is not the case, as in ETP-15 and SICStus related systems, the slow-down of systems ranking 2 (PAKCS w.r.t. $\mathcal{TOY}(\mathcal{FD}g)$)

and 3 ($\mathcal{TOY}(\mathcal{FD}s)$ w.r.t. MiniZinc) are a 25% smaller than the ones of systems ranking 1 (SICStus and Gecode). However, in ETP-21 the systems ranking 2 (again PAKCS and $\mathcal{TOY}(\mathcal{FD}g)$) and 3 (again $\mathcal{TOY}(\mathcal{FD}s)$ and MiniZinc) have a 3% and 16% (respectively) greater slow-down than the ones ranking 1.

In summary, it can be seen that ETP is a less homogeneous problem:

- The ranking of a chosen system depends on the concrete instance being run.
- The slow-down between the systems ranking 2 (respectively 3, 4 and so on) w.r.t. Gecode decreases as the instances scale up. However, as the ranking is not stable, a head-to-head comparison between the slow-down of each concrete system w.r.t. Gecode reveals that there is not a general behavior: Depending on the chosen system this slow-down can decrease, increase or decrease and then increase as the instances scale up.
- The ranking becomes more stable when it is compared by the performance order of the Gecode, ILOG Solver and SICStus related systems.
- However, in ETP-7, the performance ranking order is modified to (1) Gecode, (2) SICStus and (3) ILOG Solver. Moreover, the slow-down of the systems decrease in a 40%-80% as the systems ranking 2 and 3 (respectively) are compared.
- In ETP-15 and ETP-21, the performance ranking order remains as (1) Gecode, (2) ILOG Solver and (3) SICStus. However, in these cases there are no general patterns about how the slow-down behaves as the systems ranking 2 and 3 (respectively) are compared.

8.3 Performance Comparison of Gecode Systems

This section focuses on the Gecode related systems MiniZinc, Gecode and $\mathcal{TOY}(\mathcal{FD}g)$, analyzing first their ranking and slow-down results. Then, the instances G-11 and ETP-21 are used to discuss their search exploration (as for them it is the only factor determining the CPU time). Their search statistics results are compared, also monitoring the search in Gecode and $\mathcal{TOY}(\mathcal{FD}g)$, to justify their different results.

8.3.1 Ranking and Slow-Down Analysis

Table 8.6 presents the results of the three systems for solving the Golomb and ETP instances. As in Tables 8.2 and 8.3, the columns represent the instance, system, CPU time, slow-down and ranking, respectively. These results reveal the following conclusions.

First, the ranking is stable for, respectively, the three Golomb and ETP instances. It is (1) MiniZinc, (2) Gecode and (3) $\mathcal{TOY}(\mathcal{FD}g)$ for Golomb, and (1) Gecode, (2) $\mathcal{TOY}(\mathcal{FD}g)$ and (3) MiniZinc for ETP. As it can be seen, the performance of MiniZinc clearly varies

Instance	System	Time	Sl-Dw	Ranking
	MiniZinc	0.109	1.00	1
G-9	Gecode	0.234	2.15	2
(js)	$\mathcal{TOY}(\mathcal{FD}g)$	0.250	2.29	3
	MiniZinc	0.81	1.00	1
G-10	Gecode	2.11	2.60	2
(js)	$\mathcal{TOY}(\mathcal{FD}g)$	2.11	2.60	3
	MiniZinc	16.02	1.00	1
G-11	Gecode	41.52	2.59	2
(js)	$\mathcal{TOY}(\mathcal{FD}g)$	42.01	2.62	3
	Gecode	0.030	1.00	1
ETP-7	$\mathcal{TOY}(\mathcal{FD}g)$	0.192	6.40	2
(df)	MiniZinc	0.420	14.00	3
	Gecode	0.56	1.00	1
ETP-15	$\mathcal{TOY}(\mathcal{FD}g)$	0.90	1.61	2
(df)	MiniZinc	1.30	2.32	3
	Gecode	49.52	1.00	1
ETP-21	$\mathcal{TOY}(\mathcal{FD}g)$	50.04	1.01	2
(js)	MiniZinc	75.80	1.53	3

Table 8.6: Gecode Results

from one problem to the other. Moreover, the slow-down of Gecode (respectively $\mathcal{TOY}(\mathcal{FD}g)$) w.r.t. MiniZinc scales as the instances scale up: From the 2.15 (respectively 2.29) for G-9, to the 2.60 (respectively 2.60) for G-10 and the 2.59 (respectively 2.62) for G-11. However, in ETP, the slow-down of MiniZinc w.r.t. Gecode (respectively $\mathcal{TOY}(\mathcal{FD}g)$) nearly decreases as the instances scale up: From the 14.00 (respectively 2.18) for ETP-7, to the 2.32 (respectively 1.44) for ETP-15 and the 1.53 (respectively 1.51) for ETP-21.

In the case of Gecode and $\mathcal{TOY}(\mathcal{FD}g)$, their ranking and slow-down practically remain the same for both problems. Gecode is always behaving better than $\mathcal{TOY}(\mathcal{FD}g)$ but, whereas for *df* instances the slow-down is quite big (6.40 for ETP-7 and 1.61 for ETP-15), for the *js* instances the CPU time of $\mathcal{TOY}(\mathcal{FD}g)$ nearly matches the one of Gecode, with 1.07 for G-9 (where the 16ms difference is nearly negligible), 1.00 for G-10, 1.01 for G-11 and 1.01 for ETP-21.

These results are extremely encouraging for the use of the CFLP($\mathcal{FD}$) system $\mathcal{TOY}(\mathcal{FD}g)$, revealing that the interface from $\mathcal{TOY}(\mathcal{FD})$ to the Gecode API [48] is building up a Gecode model similar enough to the native C++ CP($\mathcal{FD}$) one, as to achieve the same performance results. Therefore, as both Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ are running (practically) the same model, the time they devote to search exploration matches. For *df* instances, the performance of $\mathcal{TOY}(\mathcal{FD}g)$ is still worse than the one of Gecode, as it

is penalized by the lazy narrowing and external solver overheads. But, for *js* instances, where this search exploration is the only factor determining the CPU time, $\mathcal{TOY}(\mathcal{FD}g)$ directly matches the CPU time of Gecode.

8.3.2 Search Exploration Analysis

This section compares the search statistics of the systems for G-11 and ETP-21. The search statistics results for G-11 are presented in Table 8.7, with their respective slowdown results presented in Table 8.8. The same information is presented for ETP-21 in Tables 8.9 and 8.10, with rows representing the search statistics of teams tt_1, tt_2 and tt_3, respectively (associated to the first tda being computed). On all these Tables, column `System` represents the system. Besides MiniZinc, Gecode and $\mathcal{TOY}(\mathcal{FD}g)$, Tables 8.9 and 8.10 add a fourth system Gecode*, representing an alternative C++ Gecode model in which both tt and $trans\_tt$ variables are represented as bi-dimensional arrays (cf. Section 7.4.2), instead of the one-dimensional ones. They also include a Column `St.`, representing the concrete team being solved. Column `Time` represents the elapsed time for performing the search exploration, and column `Cons.` the amount of constraints posted to the constraint store before the search exploration starts. Next block of three columns present some search exploration statistics, as the constraint propagations triggered (column `Propag.`), the amount of nodes/choice points explored (column `Nodes`), and the failures (column `Fails`). Finally, columns Copy and `C.T.` are devoted to the `Space::copy()` constructor, which plays a central role in the hybrid recomputation techniques the Gecode search is based on. Column Copy represents the amount of times the method is executed during the search exploration. Column `C.T.` represents the elapsed time spent in such these executions. Unfortunately, there is no way to retrieve this information in MiniZinc.

The results of tables 8.7, 8.8, 8.9 and 8.10 reveal the following conclusions:

First, column `Time` of Tables 8.7 and 8.9 show that the CPU times of MiniZinc, Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ (presented in Table 8.6) for G-11 and ETP-21 directly come from their elapsed times devoted to search exploration.

Second, the interface from $\mathcal{TOY}(\mathcal{FD})$ to Gecode allows to formulate both the Golomb and ETP problems with the same constraint network than the native C++ CP($\mathcal{FD}$) Gecode models. In the case of ETP-21, as the absences of workers differ from one team to another, the amount of constraints posted to the solver differs as well (89, 85 and 84 for the first, second and third teams, resp). In the case of MiniZinc, the algebraic CP($\mathcal{FD}$) modeling features lead to the same constraint network as in Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ in G-11, but to a greater one for ETP-21 (cf. Chapter refchapter7), ranging in 2.43-2.60 for the three teams.

However, it can be seen that there is not a direct correlation between the constraint network used and the search exploration being performed. In G-11, MiniZinc and Gecode depart from the same constraint network (cf. Table 8.7, column `Time`).

System	Time	Cons.	Propag.	Nodes	Fails	Copy	C.T.
Gecode	41.52	66	312,737,962	2,968,201	1,484,086	1,484,116	0.81
$\mathcal{TOY}(\mathcal{FD}g)$	42.01	66	320,150,379	2,968,201	1,484,086	1,484,116	0.75
MiniZinc	16.02	66	94,022,986	642,867	321,419	-	-

Table 8.7: G-11 Search Results

System	Time	Cons	Propag.	Nodes	Fails	Copy	C.T.
Gecode	1.00	1.00	1.00	1.00	1.00	1.00	1.00
$\mathcal{TOY}(\mathcal{FD}g)$	1.01	1.00	1.02	1.00	1.00	1.00	0.93
MiniZinc	0.39	1.00	0.30	0.22	0.22	-	-

Table 8.8: G-11 Slow-Down Results

System	St.	Time	Cons	Propag.	Nodes	Fails	Copy	C.T.
Gecode		2.86	89	12,801,281	398,985	194,990	194,996	0.06
Gecode*	tt_1	3.48	89	12,798,088	398,985	194,990	194,996	0.38
$\mathcal{TOY}(\mathcal{FD}g)$		2.91	89	13,030,676	398,985	194,990	194,996	0.03
MiniZinc		4.43	216	18,354,346	398,985	194,990	-	-
Gecode		19.25	85	88,453,074	2,836,539	1,418,269	1,418,271	0.54
Gecode*	tt_2	23.59	85	88,455,074	2,836,539	1,418,269	1,418,271	3.66
$\mathcal{TOY}(\mathcal{FD}g)$		19.30	85	89,411,906	2,836,539	1,418,269	1,418,271	0.22
MiniZinc		29.40	221	130,010,012	2,836,539	1,418,269	-	-
Gecode		2.55	84	11,961,872	399,669	199,834	199,836	0.11
Gecode*	tt_3	3.33	84	11,961,886	399,669	199,834	199,836	0.47
$\mathcal{TOY}(\mathcal{FD}g)$		2.58	84	12,028,347	399,669	199,834	199,836	0.05
MiniZinc		4.01	207	15,929,131	380,169	190,084	-	-

Table 8.9: ETP-21 Search Results of the Three Teams

System	St.	Time	Cons	Propag.	Nodes	Fails	Copy	C.T.
Gecode		1.00	1.00	1.00	1.00	1.00	1.00	1.00
Gecode*	tt_1	1.22	1.00	1.00	1.00	1.00	1.00	6.33
$\mathcal{TOY}(\mathcal{FD}g)$		1.02	1.00	1.02	1.00	1.00	1.00	0.50
MiniZinc		1.55	2.43	1.43	1.00	1.00	-	-
Gecode		1.00	1.00	1.00	1.00	1.00	1.00	1.00
Gecode*	tt_2	1.23	1.00	1.02	1.00	1.00	1.00	6.78
$\mathcal{TOY}(\mathcal{FD}g)$		1.00	1.00	1.01	1.00	1.00	1.00	0.41
MiniZinc		1.53	2.60	1.47	1.00	1.00	-	-
Gecode		1.00	1.00	1.00	1.00	1.00	1.00	1.00
Gecode*	tt_3	1.31	1.00	1.00	1.00	1.00	1.00	4.27
$\mathcal{TOY}(\mathcal{FD}g)$		1.01	1.00	1.01	1.00	1.00	1.00	0.45
MiniZinc		1.57	2.46	1.33	0.95	0.95	-	-

Table 8.10: ETP-21 Search Slow-Down of the Three Teams

However, due to the different propagation level for the `all_different` constraint, MiniZinc performs a search exploration containing 0.22 times the nodes and failures of the Gecode ones (cf. Table 8.8, columns `Nodes` and `Fails`). On the other hand, in the three teams of ETP-21, MiniZinc and Gecode have different constraint networks. But, whereas for the first two teams they exactly perform the same exploration (in terms of nodes and failures) for the third one they do not (cf. Table 8.10, columns `Nodes` and `Fails`).

Thus, the key factor determining the CPU time of the search exploration is the number of constraint propagations being triggered: The less constraint propagations a system triggers during the search exploration, the fastest this search is performed. In this setting, in G-11 MiniZinc triggers just a 30% of the propagations triggered by Gecode, and it achieves a slow-down of 0.39 w.r.t. it. On the other hand, for the three teams of ETP-21, MiniZinc performs a 33%-47% more propagations than Gecode, achieving a slow-down of 1.53-1.57 w.r.t. it. In the case of $\mathcal{TOY}(\mathcal{FD}g)$, it nearly matches the amount of propagations of Gecode (a 1%-2% more) for all the Golomb-11 and ETP-21 searches, and so it matches the CPU time devoted to search exploration (with a slow-down ranging in 1.01-1.02).

Third, besides the amount of propagations, in Section 7.4.2 the `copy()` constructor was identified as another key factor determining the CPU time devoted to search. On it, the importance of modeling `tt` and `trans_tt` as one-dimensional variable vectors was remarked, rather than the more intuitive bi-dimensional ones. The results of Tables 8.9 and 8.10 show this importance, presenting a system Gecode* that uses bi-dimensional arrays. This system has a small deviation of propagations w.r.t. Gecode, ranging in 1.00-1.02 (similar to $\mathcal{TOY}(\mathcal{FD}g)$). However, its slow-down w.r.t. Gecode ranges in 1.22-1.31. Looking at the times of Columns `Time` and `C.T.`, it can be seen that the differences obtained directly come from the amount of time devoted to execute the method `copy()` during the search exploration. As both systems execute the `copy()` method the same amount of times, it can be seen that the efficiency of the method in Gecode outperform the one of Gecode*.

In the case of $\mathcal{TOY}(\mathcal{FD}g)$, its interface to the Gecode API relies on a method `copy()` also based on a one-dimensional variable array. Moreover, in $\mathcal{TOY}(\mathcal{FD})$ some of the $\mathcal{FD}$ variables are saved by unifying them by the $\mathcal{H}$ solver. Thus, the time that $\mathcal{TOY}(\mathcal{FD}g)$ spends in `copy()` is even a little bit smaller than the one of Gecode (with a slow-down of 0.93 for G-11 and ranging in 0.41-0.50 for the teams of ETP-21). This is also encouraging for $\mathcal{TOY}(\mathcal{FD}g)$, that can thus reduce a little the overhead derived from the higher amount of triggered propagations, matching even more the CPU time of Gecode for performing the search exploration.

8.3.3 Monitoring the Search of Gecode and $\mathcal{TOY}(\mathcal{FD}g)$

In this section, part of the search exploration in Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ is monitored, showing that they lead to the same amount of nodes and failures, but perform a different amount of propagations.

To this end, an easier instance G-5 has been used (cf. Chapter 2), big enough as to lead to a search exploration (with the similarities and differences between Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ that want to be shown) but small enough as to make this search exploration tractable. Its optimal solution is: $m = [0, 1, 4, 9, 11]$; $d = [1, 4, 9, 11, 3, 8, 10, 5, 7, 2]$. Gist, a graphical tool of Gecode which interacts with the solver during the search exploration, has been used. It allows to monitor the execution of the search tree node by node (also executing a log method on each of them). In this case, this log method has been configured to display the domain of all the m and d variables of Golomb. The solving of G-5 in Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ lead to the same search tree exploration, which is presented in Figure 8.1. On it, the blue circles represent choice points, the green diamonds solutions that have been further improved, the yellow diamond the optimal solution, and the red squares the failures.

Top part of Figure 8.2 presents the information displayed when exploring the nodes 1, 2 and 3 of the tree, which lead to the same results in Gecode and $\mathcal{TOY}(\mathcal{FD}g)$.

Besides configuring the log method, a log constraint has been implemented, which is associated to each m and d variable. This constraint propagates each time the domain of its variable is modified, displaying its new domain. Thus, now it is also known the way the domain of the variables evolve on the execution of each node. Bottom part of Figure 8.2 presents this information for nodes 2 and 3. Column `Step` represents the step identifier, to univocally distinguish each atomic propagation pruning the domain of a variable. Columns `Gecode-NODE 2` and `TOY(FDg)-NODE 2` present the log execution of node 2 in Gecode and $\mathcal{TOY}(\mathcal{FD}g)$, respectively. It can be seen that they

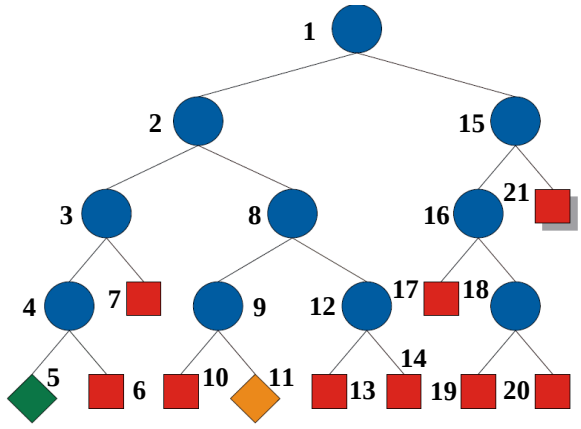

Figure 8.1: Tree of G-5 in Gist

```
          NODE 1.
          m = {0, [1..8], [3..12], [6..13], [10..15]}
          d = {[1..8], [3..12], [6..13], [10..15], [1..11],
               [3..12], [6..14], [1..10], [3..12], [2..9]}
          NODE 2.
          m = {0, 1, [3..11], [6..13], [10..15]}
          d = {1, [3..11], [6..13], [10..15], [2..10],
               [5..12], [9..14], [2..10], [3..12], [2..9]}
          NODE 3.
          m = {0, 1, 3, [7..11], [11..15]}
          d = {1, 3, [7..11], [11..15], 2,
               [6..10], [10..14], [4..8], [8..12], [4..8]}
-------------------------------------------------------------------------
Step |Gecode-NODE 2  |TOY(FDg)-NODE 2 |Gecode-NODE 3  |TOY(FDg)-NODE 3
-----|---------------|----------------|---------------|---------------
(01) |m[1] = 1       |m[1] = 1        |m[1] = 1       |m[1] = 1
(02) |d[0] = 1       |d[0] = 1        |m[2] = 3       |m[2] = 3
(03) |d[5] = [5..12] |d[5] = [5..12]  |d[0] = 1       |d[0] = 1
(04) |d[4] = [2..11] |d[4] = [2..11]  |d[1] = 3       |d[1] = 3
(05) |d[6] = [9..14] |d[6] = [9..14]  |d[5] = [5..12] |d[5] = [5..12]
(06) |d[7] = [2..10] |d[7] = [2..10]  |d[4] = 2       |d[4] = 2
(07) |m[2] = [3..11] |m[2] = [3..11]  |d[6] = [9..14] |d[6] = [9..14]
(08) |m[2] = [3..11] |d[4] = [2..10]  |d[7] = [3..10] |d[7] = [3..10]
(09) |d[4] = [2..10] |m[2] = [3..11]  |d[8] = [7..12] |d[8] = [7..12]
(10) |               |                |d[7] = [4..10] |d[7] = [4..10]
(11) |               |                |d[9] = [4..9]  |d[9] = [4..9]
(12) |               |                |m[3] = [7..13] |m[3] = [7..13]
(13) |               |                |d[2] = [7..13] |d[5] = [6..12]
(14) |               |                |d[9] = [4..8]  |d[2] = [7..13]
(15) |               |                |m[3] = [7..11] |m[3] = [7..11]
(16) |               |                |m[4] = [11..15]|d[9] = [4..8]
(17) |               |                |d[2] = [7..11] |m[4] = [11..15]
(18) |               |                |d[3] = [11..15]|d[5] = [6..10]
(19) |               |                |d[5] = [6..10] |d[2] = [7..11]
(20) |               |                |d[7] = [4..8]  |d[7] = [4..8]
(21) |               |                |d[8] = [8..12] |d[8] = [8..12]
(22) |               |                |d[6] = [10..14]|d[6] = [10..14]
(23) |               |                |               |d[3] = [11..15]
```

Figure 8.2: Variable Domains Prunings for Nodes 1, 2 and 3

basically follow the same steps, although the order of execution of steps 8 and 9 (pruning the domains of m_2 and d_4) are interchanged. Finally, columns `Gecode-NODE 3` and `TOY(FDg)-NODE 3` present the log execution of node 3. Again, it can be seen that the propagation of some variable domains are interchanged from Gecode to $\mathcal{TOY}(\mathcal{FD}g)$. But, most importantly, to achieve the propagation of `d[5]` $\mathcal{TOY}(\mathcal{FD}g)$ performs one more step than Gecode (step 13).

This example represents in detail what is happening in Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ in Tables 8.7 and 8.9, where it can be seen that, even containing the same constraint network and performing the same search exploration (in terms of nodes and failures) they lead to different number of constraint propagations.

8.4 Performance Comparison of ILOG Solver Systems

This section is similar to Section 8.3, but it is focused on the ILOG Solver related systems ILOG OPL, ILOG Solver and $\mathcal{TOY}(\mathcal{FD}i)$.

8.4.1 Ranking and Slow-Down Analysis

Table 8.11 presents the results of the three systems for solving the Golomb and ETP instances. As in Table 8.6, the columns represent the instance, system, CPU time, slow-down and ranking, respectively. These results reveal the following conclusions:

First, the ranking for Golomb instances is (1) $\mathcal{TOY}(\mathcal{FD}i)$, (2) ILOG Solver and (3) ILOG OPL. For the *df* instances ETP-7 and ETP-15, the ranking is (1) ILOG Solver, (2) ILOG OPL and (3) $\mathcal{TOY}(\mathcal{FD}i)$, but for the *js* instance ETP-21, the ranking is (1) ILOG Solver, (2) $\mathcal{TOY}(\mathcal{FD}i)$ and (3) ILOG OPL.

The most two remarkable observations are that $\mathcal{TOY}(\mathcal{FD}i)$ is the fastest system for Golomb (behaving even better than ILOG Solver), and that, for ETP, the ranking is dependent of dealing with a *df* or a *js* instance. Besides that, there are some similarities w.r.t. the Gecode related systems ranking: The ranking is stable for the Golomb instances and it varies from the Golomb to the ETP problem. The C++ CP($\mathcal{FD}$) model ranks 2 for the Golomb instances, and 1 for the ETP ones. The system ranking 1 for the Golomb instances behaves as the worst for the ETP ones.

Second, although the slow-down interval for *df* instances is quite big, ranging in 1.04-8.85, the one for *js* ones is quite narrow, ranging in 1.06-1.33. This reveals that, besides having a less homogeneous ranking, the general differences among the ILOG Solver related systems are smaller than the ones achieved among the Gecode systems. Specifically, whereas the slow-down of ILOG Solver w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ for Golomb instances decreases as the instances scale up (from 1.11 for G-9 to the 1.08 and 1.06 for G-10 and G-11, respectively), the slow-down of $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. ILOG Solver for ETP-21 is 1.17.

Instance	System	Time	Sl-Dw	Ranking
G-9	$\mathcal{TOY}(\mathcal{FD}i)$	0.421	1.00	1
	ILOG Solver	0.468	1.11	2
	ILOG OPL	0.500	1.19	3
G-10	$\mathcal{TOY}(\mathcal{FD}i)$	3.56	1.00	1
	ILOG Solver	3.85	1.08	2
	ILOG OPL	4.40	1.24	3
G-11	$\mathcal{TOY}(\mathcal{FD}i)$	72.65	1.00	1
	ILOG Solver	77.08	1.06	2
	ILOG OPL	88.28	1.22	3
ETP-7	ILOG Solver	0.156	1.00	1
	ILOG OPL	0.360	2.31	2
	$\mathcal{TOY}(\mathcal{FD}i)$	1.380	8.85	3
ETP-15	ILOG Solver	0.96	1.00	1
	ILOG OPL	1.00	1.04	2
	$\mathcal{TOY}(\mathcal{FD}i)$	3.28	3.42	3
ETP-21	ILOG Solver	93.42	1.00	1
	$\mathcal{TOY}(\mathcal{FD}i)$	109.02	1.17	2
	ILOG OPL	124.70	1.33	3

Table 8.11: ILOG Results

These results are twofold regarding the interface from $\mathcal{TOY}(\mathcal{FD})$ to ILOG Solver. On the one hand, the overheads of $\mathcal{TOY}(\mathcal{FD}i)$ for solving *df* instances are much greater than the ones of ILOG Solver, and even slightly greater than the ones of $\mathcal{TOY}(\mathcal{FD}g)$ (cf. Table 5.2, column `Perc_I`). This directly points at the interface from $\mathcal{TOY}(\mathcal{FD})$ to ILOG Solver, revealing that $\mathcal{TOY}(\mathcal{FD}i)$ needs more management of ILOG Concert objects than the native model of ILOG Solver (e.g., each of the tasks proposed in Section 3.2). Thus, the slow-down of $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. ILOG Solver for ETP-7 and ETP-15 results to be quite big.

On the other hand, these same results present $\mathcal{TOY}(\mathcal{FD}i)$ as a very competitive system for solving *js* instances. Moreover, differently to what happens with Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ (that nearly match their CPU times), these results reveal that, depending on the problem being solved, $\mathcal{TOY}(\mathcal{FD}i)$ can behave either better (and also worse) than ILOG Solver. This points again at the $\mathcal{TOY}(\mathcal{FD})$ interface to ILOG Solver, revealing that there is a mismatch between the model built up by $\mathcal{TOY}(\mathcal{FD}i)$ and the one built up by the C++ CP($\mathcal{FD}$) model. Next section discusses the different management required by ILOG Solver, which leads to a different amount of variables and constraints posted to the solver. As ILOG Solver and $\mathcal{TOY}(\mathcal{FD}i)$ are running (partially) different constraint networks, the time they spend in search exploration differs. Finally, in Section 8.4.3 the execution of these Golomb and ETP models are monitored, identifying the cases in

which $\mathcal{TOY}(\mathcal{FD}i)$ saves variables and constraints w.r.t. ILOG Solver, and viceversa.

8.4.2 Search Exploration Analysis

This section is similar to Section 8.3.2, but with tables 8.12 and 8.13 (respectively 8.14 and 8.15) representing the search statistics and slow-down for G-11 (respectively ETP-21) in ILOG Solver, $\mathcal{TOY}(\mathcal{FD}i)$ and ILOG OPL. As a difference with the Gecode search results previously displayed, columns `Vars` and `Cons` represent the amount of variables and constraints (respectively) posted to the constraint store before the search exploration starts. But, unfortunately, neither the API of ILOG OPL nor the one of ILOG Solver provide a method to compute the amount of constraint propagations triggered during the search exploration. As it was seen, this was the key factor determining the elapsed time performed by the Gecode systems, so the lack of this information supposes a drawback now for the analysis among the ILOG Solver systems.

Focusing then on the nodes, failures, variables and constraints presented in tables 8.7-8.10 the following conclusions are obtained:

First, as in Gecode systems, column `Time` of tables 8.12 and 8.14 show that the CPU times of ILOG OPL, ILOG Solver and $\mathcal{TOY}(\mathcal{FD}i)$ (presented in Table 8.11) for G-11 and ETP-21 directly come from their elapsed time devoted to tackle the search exploration.

Second, interestingly, both in G-11 and ETP-21 the three systems use the same amount of nodes and failures (in ETP-21 the differences are nearly negligible, not reaching even a 1%). This represents a difference w.r.t. Table 8.7, where the algebraic CP($\mathcal{FD}$) system MiniZinc used less nodes and failures than Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ for solving G-11. The formulation of Golomb is as simple that both MiniZinc and ILOG OPL models are exactly the same (besides the small grammar differences between both languages). This formulation includes as its unique global constraint the `all_different` among d variables. For it, ILOG OPL supports value consistency (as Gecode, ILOG Solver, $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$), but MiniZinc does not. Thus, it is the factor making MiniZinc to perform a smaller search exploration (in terms of nodes, failures and constraint propagations).

Besides obtaining the same statistics after exploring the search tree, columns `Vars` and `Cons` of tables 8.12 and 8.14 reveal that the three related systems use a different amount of variables and constraints. These represent a difference w.r.t. Gecode, where only MiniZinc used a higher amount of constraints for ETP-21 (in all cases with no available info about the amount of variables used).

ILOG OPL uses more variables and constraints than ILOG Solver for ETP-21 and G-11. But, when comparing ILOG Solver and $\mathcal{TOY}(\mathcal{FD}i)$, the difference of variables and constraints depends on the problem: Whereas for G-11 $\mathcal{TOY}(\mathcal{FD}i)$ uses less variables and constraints than ILOG Solver, for ETP-21 it uses more. This amount of variables and constraints turn to be the key factor determining the elapsed time, as they represent the constraint network the solver has to deal with. As it can be seen, the smaller this

System	Time	Nodes	Fails	Vars	Cons
TOY(FDi)	72.65	1,484,102	1,484,086	55	56
ILOG Solver	77.08	1,484,102	1,484,086	66	67
ILOG OPL	88.28	1,484,102	1,484,088	76	132

Table 8.12: G-11 Search Results

System	Time	Nodes	Fails	Vars	Cons
TOY(FDi)	1.00	1.00	1.00	1.00	1.00
ILOG Solver	1.06	1.00	1.00	1.20	1.20
ILOG OPL	1.22	1.00	1.00	1.38	2.36

Table 8.13: G-11 Slow-Down Results

System	St.	Time	Nodes	Fails	Vars	Cons
ILOG Solver		4.93	194,992	194,990	141	99
TOY(FDi)	tt_1	5.91	194,992	194,990	186	104
ILOG OPL		6.30	195,013	195,011	147	183
ILOG Solver		37.29	1,418,269	1,418,269	141	100
TOY(FDi)	tt_2	42.98	1,418,269	1,418,269	181	105
ILOG OPL		51.08	1,418,276	1,418,276	147	183
ILOG Solver		4.45	190,834	190,834	141	99
TOY(FDi)	tt_3	4.88	190,834	190,834	191	104
ILOG OPL		4.72	190,091	190,091	147	183

Table 8.14: ETP-21 Search Results of the Three Teams

System	St.	Time	Nodes	Fails	Vars	Cons
ILOG Solver		1.00	1.00	1.00	1.00	1.00
TOY(FDi)	tt_1	1.20	1.00	1.00	1.32	1.05
ILOG OPL		1.28	1.00	1.00	1.04	1.85
ILOG Solver		1.00	1.00	1.00	1.00	1.00
TOY(FDi)	tt_2	1.15	1.00	1.00	1.28	1.05
ILOG OPL		1.37	1.00	1.00	1.04	1.83
ILOG Solver		1.00	1.00	1.00	1.00	1.00
TOY(FDi)	tt_3	1.10	1.00	1.00	1.35	1.05
ILOG OPL		1.06	1.00	1.00	1.04	1.83

Table 8.15: ETP-21 Search Slow-Down of the Three Teams

network is, the faster the solver performs the search.

In G-11, it can be seen that, even with such an easy formulation, ILOG Solver uses 20% more variables and constraints than $\mathcal{TOY}(\mathcal{FD}i)$, leading to a slow-down of 1.06. In the case of ILOG OPL, its formulation leads to a 38% more variables and 136% more constraints than $\mathcal{TOY}(\mathcal{FD}i)$, leading to a slow-down of 1.22.

In ETP-21, it can be seen that ILOG Solver is the system with less variables and constraints for the 3 team subproblems, and thus it is the fastest one for solving all of them. However, there is not a clear pattern between the constraint network and the slow-down. In the case of $\mathcal{TOY}(\mathcal{FD}i)$, the slow-down of constraints is stable (1.32, 1.28 and 1.35, respectively). However, the slow-down achieved decreases (1.20, 1.15 and 1.10, respectively), not following a relation w.r.t. the amount of variables posted to the solver. In the case of ILOG OPL, both the slow-downs of variables and constraints are stable (at 1.04 and 1.83, respectively), but the slow-downs are quite different (1.28, 1.37 and 1.06, respectively) Finally, a similar situation happens when comparing $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. ILOG OPL. For the 3 teams, ILOG OPL uses around 20% less variables than $\mathcal{TOY}(\mathcal{FD}i)$, but around 75% more constraints. However, the slow-down of ILOG OPL w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ does not follow a clear pattern (1.07, 1.19 and 0.97, respectively), being even faster for the third team.

8.4.3 Monitoring the Execution of ILOG Solver and $\mathcal{TOY}(\mathcal{FD}i)$

This section monitors the execution of the ILOG Solver and $\mathcal{TOY}(\mathcal{FD}i)$ models, justifying the different amount of variables and constraints they post to the solver (something that did not happen with Gecode and $\mathcal{TOY}(\mathcal{FD}g)$).

The interfaces from $\mathcal{TOY}(\mathcal{FD})$ to Gecode and from $\mathcal{TOY}(\mathcal{FD})$ to ILOG only differ in their C++ code, i.e., in the management of the C++ API objects of the concrete library being used. In this setting, the results show that, whereas $\mathcal{TOY}(\mathcal{FD}g)$ is building up the same Gecode model as the one built up when programming directly in the C++ framework, $\mathcal{TOY}(\mathcal{FD}i)$ is not doing so. That is, even following the same problem formulation, $\mathcal{TOY}(\mathcal{FD}i)$ and ILOG Solver are leading to different constraint networks.

Whereas for G-11 $\mathcal{TOY}(\mathcal{FD}i)$ builds up a model with less variables and constraints than the one built up by ILOG Solver, for ETP-21 the situation is the opposite. This shows that there are cases in which the interface from $\mathcal{TOY}(\mathcal{FD})$ to ILOG Concert and ILOG Solver allows to save variables and constraints w.r.t. the direct modeling in C++, and viceversa. By using `printInformation()`, the amount of variables and constraints that $\mathcal{TOY}(\mathcal{FD}i)$ and ILOG Solver give rise on each execution step of their programs is measured. In this section, the differences for both G-11 and ETP-21 are presented, analyzing where do they come from.

G-11. Table 8.12 shows that, whereas ILOG Solver requires 66 variables and 67 constraints, $\mathcal{TOY}(\mathcal{FD}i)$ requires 55 and 56, respectively. This difference of 11 variables and constraints come from 2 situations:

First, in $\mathcal{TOY}(\mathcal{FD}i)$ m_0 is unified to 0 by the $\mathcal{H}$ solver. This allows $\mathcal{TOY}(\mathcal{FD}i)$ to save both the variable and the constraint. However, in ILOG Solver both the variable and the constraint are taken into account by the $\mathcal{FD}$ solver (cf. Section 7.1.2).

Second, the first 10 variables of $d_0, \ldots, d_9$ represent the subtractions of $m_1, \ldots, m_{10}$ and m_0. In $\mathcal{TOY}(\mathcal{FD}i)$, these constraints are managed by the lazy narrowing as `D0 == M1 #- M0`, ..., `D9 == M10 #- M0`, where `D0`...`D9` appear for first time in the program. Their management is performed in two steps: `Mi #- 0 = Aux` and (ii) `Di == Aux`. In the former, the interface of ILOG Concert and ILOG Solver treats the constraint as a basic one, where the new `IloIntVar aux` is set to be equal to the subtraction of the already existing `IloIntVar` (`m[i]`) and the constant 0. As the subtraction is a dummy one, ILOG Concert makes `aux` and `mi` to be two different `IloIntVar` representations of the same implementation variable (cf. Section 7.4.2). Thus, the constraint solver saves the variable (aux) and the constraint (`aux = mi - 0`). Then, for the management of `Di == Aux` in $\mathcal{TOY}(\mathcal{FD}i)$, as `Di` was not previously identified as an $\mathcal{FD}$ variable, the unification with Aux is in charge of the $\mathcal{H}$ solver. For the rest of the program, each occurrence of `Di` (as in the `alldifferent` constraint) will be identified with Aux in $\mathcal{TOY}(\mathcal{FD}i)$ and with aux in the C++ code of the interface.

However, in the ILOG Solver model, `d` is initialized as an `IloIntVarArray` with an initial domain. Thus, the constraint solver considers `d[0]`...`d[9]` as variables and each `d[i] == m[i] - m[0]` as a constraint.

ETP-21. Table 8.14 shows that, whereas for the first team ILOG Solver requires 141 variables and 99 constraints, $\mathcal{TOY}(\mathcal{FD}i)$ requires 186 and 104, respectively. This difference comes from quite a few situations:

First, in `p_tt_ta_5` (cf. Section 5.3.1) the implication constraints `C1 => C2` are managed as `IloIfThen` by the ILOG Concert API. Any implication constraint occurring in the ILOG Solver model leads to two constraints. However, in $\mathcal{TOY}(\mathcal{FD}i)$, the managing of the interface can save some constraints: In `D1 (#=) 2 #=> A1 (#=) 3`, as the domain of `A1` is `0..2`, the solver only posts one constraint, i.e., `D1 /= 2`, making the domain of `D1` to be `1 \/ 3`. Also, in `D1 (#=) 3 #=> A1 (#=) 3`, the solver only posts the constraint `D1 /= 3`. This bounds the domain of `D1` to 1, also binding `D4`, `D7`, ..., `D19` to 1, and pruning the value 1 from the domain of `D2`, `D3`, ..., `D20` and `D21`. Then, in `D2 (#=) 1 #=> A2 (#=) 0`, as the domain of `D2` is `2..3`, the expression is trivially entailed and no constraints are posted to the solver.

Second, consider the following $\mathcal{TOY}(\mathcal{FD}i)$ goal: `domain [X,Y] 1 2, X #> 1, sum [X,Y] (#=) Z`. By using incremental propagation, when the goal reaches the last constraint, the variable `X` is bound to 2, so the system evaluates `sum [2,Y] (#=) Z`. Then, the interface of $\mathcal{TOY}(\mathcal{FD}i)$ uses `Y` to find out the mate `IloIntVar` in the C++ code. Unfortunately, for `X` the only information that can be inferred is that there must be a decision variable bound to 2, but not which one is it. The API of ILOG Concert does not allow to use an integer `IloInt` as an argument of a `sum` constraint, but only `IloIntVars`. In that context, an extra dummy `IloIntVar` must be created, whose

domain is initially bound to 2. The solver considers this variable as part of the constraint network, but at least it does not consider its domain initialization as an additional constraint. This situation is produced in $\mathcal{TOY}(\mathcal{FD}i)$ with any bound variable involved on a `sum`, `distribute`, `count` and `alldifferent` constraint, implying the use of more variables.

In the ILOG Solver model, the sum constraint can be easily modelled over x (even if it is bound at that moment) and y, so no additional variable is necessary.

Third, for managing a `sum List Op R`, $\mathcal{TOY}(\mathcal{FD}i)$ creates one variable per integer of `List`, one variable for `Val`, one constraint for `sum List #= Val`, and one constraint for `Val Op R`. In the case of ILOG Solver, all these variables and constraints are skipped. For managing a `distribute`, $\mathcal{TOY}(\mathcal{FD}i)$ creates one variable per integer of `cards` and one variable per integer of `vars`. In the case of ILOG Solver, it skips the extra variables of `vars`. For managing a `count L List Op R`, $\mathcal{TOY}(\mathcal{FD}i)$ creates one variable for `L`, one variable per integer of `List`, one variable for `Val`, one constraint for `card(L) in List #= Val`, and one constraint for `Val Op R`. In the case of ILOG Solver, it skips the variables of `L` and `List`.

Fourth, the use of complex constraints is supported in $\mathcal{TOY}(\mathcal{FD}i)$, but they are internally decomposed into multiple basic constraints, and thus extra variables are required to represent them. For example, the expression `X #+ 2 #* Y` requests first to evaluate `2 #* Y`, generating an extra variable Z. Then, the sum `X #+ Z` is applied. In the case of ILOG Solver the additional variables are skipped, and the whole expression is represented as a single constraint.

So, as it can be seen, there are cases in which $\mathcal{TOY}(\mathcal{FD}i)$ saves variables and constraints w.r.t. ILOG Solver, and viceversa. Thus, depending on the formulation of the problem, more specifically depending on the amount of these situations that arise in this formulation, either $\mathcal{TOY}(\mathcal{FD}i)$ or ILOG Solver will lead to a smaller constraint network, obtaining thus a better performance.

8.5 Performance Comparison of SICStus Systems

This section is similar to Section 8.3, but it is focused on the SICStus `clpfd` related systems SICStus, PAKCS and $\mathcal{TOY}(\mathcal{FD}s)$.

8.5.1 Ranking and Slow-Down Analysis

Table 8.16 presents the results of the three systems for solving the Golomb and ETP instances. As in Table 8.6, the columns represent the instance, system, CPU time, slowdown and ranking, respectively. These results reveal the following conclusions:

First, the ranking is totally stable for all the instances, being (1) SICStus, (2) PAKCS and (3) $\mathcal{TOY}(\mathcal{FD}s)$. This stability was not achieved neither in Gecode nor in ILOG

Instance	System	Time	Sl-Dw	Ranking
	SICStus	0.764	1.00	1
G-9	PAKCS	0.810	1.06	2
(js)	$\mathcal{TOY}(\mathcal{FDS})$	0.842	1.10	3
	SICStus	6.65	1.00	1
G-10	PAKCS	7.12	1.07	2
(js)	$\mathcal{TOY}(\mathcal{FDS})$	7.35	1.11	3
	SICStus	143.60	1.00	1
G-11	PAKCS	151.60	1.06	2
(js)	$\mathcal{TOY}(\mathcal{FDS})$	153.05	1.07	3
	SICStus	0.094	1.00	1
ETP-7	PAKCS	0.220	2.34	2
(df)	$\mathcal{TOY}(\mathcal{FDS})$	0.248	2.64	3
	SICStus	1.88	1.00	1
ETP-15	PAKCS	2.36	1.26	2
(df)	$\mathcal{TOY}(\mathcal{FDS})$	3.22	1.71	3
	SICStus	190.48	1.00	1
ETP-21	PAKCS	199.30	1.05	2
(js)	$\mathcal{TOY}(\mathcal{FDS})$	338.16	1.78	3

Table 8.16: SICStus Results

Solver, and reveals that the overheads of (respectively) SLD resolution and lazy narrowing do not distort the performance order of the systems for solving *df* instances.

However, even in this context, the slow-down of PAKCS and $\mathcal{TOY}(\mathcal{FDS})$ w.r.t. SICStus ranges in 1.05-1.11 (with a 1.78 exception for $\mathcal{TOY}(\mathcal{FDS})$ in ETP-21, which will be discussed later). For *df* instances, this slow-down ranges in 1.26-2.64, representing a much worse performance of the CFLP($\mathcal{FD}$) systems. More specifically, for *js* instances, the slow-down of PAKCS w.r.t. SICStus is quite stable, ranging in 1.05-1.07. Similarly, the slow-down of $\mathcal{TOY}(\mathcal{FDS})$ w.r.t. SICStus for solving the Golomb instances ranges in 1.07-1.11. But, for ETP-21, the slow-down of $\mathcal{TOY}(\mathcal{FDS})$ w.r.t. SICStus increases until 1.78 (in comparison with the 1.05 of PAKCS). This result is less encouraging for the use of $\mathcal{TOY}(\mathcal{FDS})$, as it is even worse than the one achieved for the *df* instance ETP-15 (1.71), and not much smaller than the one for ETP-7 (2.64), where the impact of the additional host overhead is more important. In this setting, in $\mathcal{TOY}(\mathcal{FDS})$ it can be seen that, more than a different behavior between *js* and *df* instances, there is a different behavior between Golomb and ETP instances. This can also be observed when comparing $\mathcal{TOY}(\mathcal{FDS})$ with PAKCS, as in all the instances PAKCS is faster but, whereas for Golomb instances the slow-down decreases as the instances scale up (1.04 for G-9, 1.03 for G-10 and 1.01 for G-11), for ETP instances the slow-down increases as the instances scale up (1.13 for ETP-7, 1.36 for ETP-15 and 1.70 for ETP-21).

8.5.2 Search Exploration Analysis

This section is similar to Section 8.3.2, but representing the search statistics for SICStus and $\mathcal{TOY}(\mathcal{FD}s)$ (PAKCS provides no option/command to obtain any information regarding the constraint solver statistics). The analysis includes the CPU time for performing the search exploration, and the results displayed by the predicate `fd_statistics`, which include the amount of:

- Constraints posted to the constraint store before the search exploration starts. Each of these primitive constraints is translated by SICStus `clpfd` into an indexical (cf. Section 2.3.3), which is the one being definitely posted to the solver.
- Resumptions performed during the search exploration. Each resumption represents a sort of constraint propagation, i.e., the triggering of the filtering algorithm of an indexical, evaluating (and possibly pruning) the domain of its variables involved.
- Entailments performed during the search exploration (an indexical is entailed when the relation among its variables become trivially true).
- Prunings of variables during the search exploration.
- Backtracks (or failures nodes) reached during the search exploration.

Before presenting the search results of SICStus and $\mathcal{TOY}(\mathcal{FD}s)$, Figure 8.3 presents an example illustrating the behavior of SICStus `clpfd` w.r.t. the indexical generated from a constraint primitive, as well as the constraint solver statistics from propagating it: The predicate `sub(A,B,C):- A - B #= C`, has a single clause, posting a subtraction constraint to `clpfd`. Figure 8.3 presents a SICStus session solving two goals, which create an $\mathcal{FD}$ variable X (with domain values 20..22), post Y to be equal to the subtraction of X and 10, display the constraint solver statistics after propagating the constraint, and requests also the domain size of Y. Both goals are equivalent but, whereas the first one posts the subtraction constraint directly, the second one posts it by using the predicate `sub/3`. However, there are some differences in their results:

First, the domain of Y in the first goal is `{10}\/{12}` (with size `SY = 2`), but in the second goal is `10..12` (with size `SY = 3`). This reveals that, whereas the indexical generated by `clpfd` from the constraint `X - 10 #= Y` maintains domain-consistency, the indexical generated from `sub(X,10,Y)` maintains bound-consistency. Unfortunately, the propagation level of the indexical a primitive constraint generates is not configurable in SICStus `clpfd`. This example is relevant in the context of the comparison between SICStus and $\mathcal{TOY}(\mathcal{FD}s)$, as whereas the SICStus model posts the constraints as the first goal proposed, $\mathcal{TOY}(\mathcal{FD}s)$ does it as the second one. Thus, both the propagation they achieve, and the time they spent in running this propagation algorithm is different.

```
| ?- domain([X],20,22), X #\= 21, X - 10 #= Y, fd_statistics, fd_size(Y,SY).
Resumptions: 2; Entailments: 0; Prunings: 3; Backtracks: 0; Constraints: 1
SY = 2,
X in{20}\/{22},
Y in{10}\/{12} ? ;
no
| ?- domain([X],20,22), X #\= 21, sub(X,10,Y), fd_statistics, fd_size(Y,SY).
Resumptions: 4; Entailments: 0; Prunings: 3; Backtracks: 0; Constraints: 1
SY = 3,
X in{20}\/{22},
Y in 10..12 ? ;
no
```

Figure 8.3: SICStus Different Behavior for Equivalent Goals

Furthermore, these differences in the propagation algorithms being executed are not reflected in the constraint solver statistics, as only the amount of indexicals being generated is displayed. The first goal prunes one variable domain value more than the second goal (the value 11 of Y). However, the statistics says that both goals execute 3 prunings. This reflects that `fd_statistics` just counts the prunings in the lower and upper bound values of the variables (and thus the value 11 of Y is not counted for the statistics). As the goals just contains one indexical, it can be ensured that the resumption statistics represent the amount of times the propagation algorithm of this indexical is being executed. Whereas the first goal executes twice the domain-consistency algorithm, the second goal executes four times the bound-consistency algorithm. However, for a same indexical, executing a domain-consistency filtering algorithm takes longer than executing a bound-consistency one. Thus, a direct correlation cannot be found between the amount of resumptions performed and the CPU time for executing these resumption filtering algorithms.

Tables 8.17 and 8.18 represent the search statistics (and slow-down) of SICStus and $\mathcal{TOY}(\mathcal{FD}s)$ for Golomb and ETP (respectively), revealing the following conclusions:

First, column `Time` of both tables show that the CPU times of SICStus and $\mathcal{TOY}(\mathcal{FD}s)$ (presented in Table 8.16) for G-11 and ETP-21 directly come from their CPU times devoted for the search.

Second, $\mathcal{TOY}(\mathcal{FD}s)$ generates the same amount of indexicals than SICStus for G-11 and each of the 3 teams of ETP-21. Moreover, by using `fd_statistics` it has been ensured that, for each execution step of the SICStus and $\mathcal{TOY}(\mathcal{FD}s)$ programs, they match the amount of indexicals posted to the solver. However, as it happened in Figure 8.3, the consistency algorithm of the indexicals generated by `clpfd` differ in both executions.

Regarding the other search statistics, the amount of backtracks of $\mathcal{TOY}(\mathcal{FD}s)$ and SICStus matches, but this is not noticeable, as it also happened for $\mathcal{TOY}(\mathcal{FD}g)$ and

System	Time	Cons	Resump.	Entail.	Prun.	Backt.
SICStus	143.60	56	645,967,067	26,192,797	207,110,715	1,484,088
$\mathcal{TOY}(\mathcal{FD}s)$	153.05	56	593,495,878	26,385,042	207,068,902	1,484,088
T/S	1.07	1.00	0.92	1.01	1.00	1.00

Table 8.17: G-11 Search Results

System	St.	Time	Indexicals	Resump.	Entail.	Prun.	Backt.
SICStus		9.33	231	18,663,982	8,381,326	11,619,779	194,990
$\mathcal{TOY}(\mathcal{FD}s)$	tt_1	18.61	231	18,953,619	8,381,326	11,619,779	194,990
T/S		1.99	1.00	1.02	1.00	1.00	1.00
SICStus		78.27	224	110,665,909	62,977,336	76,755,493	1,418,269
$\mathcal{TOY}(\mathcal{FD}s)$	tt_2	136.11	224	113,707,230	62,977,336	76,755,493	1,418,269
T/S		1.74	1.00	1.03	1.00	1.00	1.00
SICStus		7.61	225	16,281,460	7,688,457	10,326,448	199,834
$\mathcal{TOY}(\mathcal{FD}s)$	tt_3	14.14	225	16,617,391	7,688,457	10,326,448	199,834
T/S		1.86	1.00	1.02	1.00	1.00	1.00

Table 8.18: ETP-21 Search Results of the Three Teams

Gecode (respectively for $\mathcal{TOY}(\mathcal{FD}i)$ and ILOG Solver). Regarding the amount of prunings and entailments of G-11, it can be seen that $\mathcal{TOY}(\mathcal{FD}s)$ needs less variable domains prunings to entail more constraints. It seems to be counter intuitive, but, as it was seen in the example of Figure 8.3, the pruning results only take into account the lower and upper bound prunings. In the case of ETP-21, the amount of entailments and prunings directly matches.

Finally, the amount of resumptions could be the key factor determining the CPU time, as it indicates the amount of filtering algorithms being executed during search. However, these results are not relevant by themselves, but only jointly with the CPU time required by the filtering algorithm of each indexical, and the amount of times each indexical is executed. For example, in G-11 $\mathcal{TOY}(\mathcal{FD}s)$ spends 10 seconds more (a slow-down of 1.07 w.r.t SICStus) in performing 52 millions of executions less, so the ratio of time per filtering algorithm execution must be higher.

8.5.3 Constraint Network of SICStus and $\mathcal{TOY}(\mathcal{FD}s)$

This section monitors the set of indexicals (constraint network) that SICStus and $\mathcal{TOY}(\mathcal{FD}s)$ generate for the instance G-5 (for which the variables and constraints were presented in Section 2.1), showing that it is different in both systems. The predicate `call_residue` is used, which prints the set of variables (with their current domain) and each indexical posted to the constraint store. It is applied just before triggering the search exploration, obtaining thus the content of the store once the whole problem formulation has been posted.

Top and bottom parts of Figure 8.4 present the results for `call_residue` in SICStus and $\mathcal{TOY}(\mathcal{FD}s)$, respectively. The differences between them are presented in boldface. For each variable (respectively indexical), the format used by `call_residue` presents first the set of variables involved and then the domain (respectively constraint associated). Each element is univocally identified.

Regarding the SICStus session (top part of Figure 8.4) the m and d variables are represented in lines 12..21. As it was pointed out in Section 8.4.3, m_0 is unified to 0 by the $\mathcal{H}$ solver, and thus the constraints $d_0 = m_1 - 0$, ..., $d_3 = m_4 - 0$ turn to be trivially solved by the $\mathcal{H}$ unifications of d_0 with m_1, ..., d_3 with m_4. Thus, the variable m_0 does not belong to the constraint store. The variable _8074 of line (12) represents d_0 and m_1. The variable _8075 of line (13) represents d_1 and m_2. The variable _8076 of line (14) represents d_2 and m_3. The variable of _8077 of line (15) represents d_3 and m_4. The variables _8071, _8072, _8073, _8069, _8070 and _8068 of lines (16)...(21) represent $d_4 \ldots d_9$, respectively.

The constraint $m_0 < m_1$ is in fact $0 < m_1$, which is understood as a trivial constraint by `clpfd` (pruning the lower bound of m_1). Thus, it is not included in the constraint store. The lines (02), (03) and (04) represent the constraints $m_1 < m_2$, $m_2 < m_3$ and $m_3 < m_4$, respectively. These constraints are translated to indexicals following the pattern `t=<u+c`, being represented as $m_1 <= m_2 - 1$, $m_2 <= m_3 - 1$ and $m_3 <= m_4 - 1$ (respectively).

Once again, the constraints setting $d_0, \ldots, d_3$ to the subtraction of $m_1, \ldots, m_4$ and m_0 (i.e., the constant 0) are managed by the $\mathcal{H}$ solver. The lines (06)..(11) constrain $d_4, \ldots, d_9$. For example, line (06) constrains $d_4 = m_2 - m_1$ and line (11) constrains $d_9 = m_4 - m_3$. On each case, the constraints are translated into indexicals following the pattern `x+y=t`. The constraints of lines (06) and (11) are represented as $d_4 + m_1 = m_2$ and $d_9 + m_3 = m_4$ (respectively).

The line (05) represents the symmetry breaking constraint $d_0 < d_9$, which is translated into an indexical following the pattern `t>=u+c`. The constraint is represented as $d_9 >= d_0 + 1$. Finally, the line (01) represents the $alldifferent$ constraint.

Regarding the $\mathcal{TOY}(\mathcal{FD}s)$ session (bottom part of Figure 8.4) the differences found (w.r.t. SICStus) are the following:

First, the variable identifiers are much higher than the ones for SICStus (with values over _30,000 in contrast to the _8,000 of SICStus). These identifiers have a continue increasing order from _30421 to _30430, in contrast with the intermittent increasing order of SICStus: _8074,...,_8077, then _8071,...,_8073, then _8069,...,_8070 and finally _8068. This reveals that the load of the $\mathcal{TOY}(\mathcal{FD})$ libraries on top of SICStus (to launch a $\mathcal{TOY}(\mathcal{FD}s)$ session) leaves a smaller amount of free memory to further tackling the labeling execution.

Second, the lines (02), (03) and (04) represent the constraints $m_1 < m_2$, $m_2 < m_3$ and $m_3 < m_4$, respectively. These constraints are translated into indexicals following the pattern `t>=u+c`. The constraints are represented as $m_2 >= m_1 + 1$, $m_3 >= m_2 + 1$

```
————————————————————————————————————————
| ?- call\_residue((golomb(5,true,M)),Res), write(Res).
————————————————————————————————————————
(01) [[_8074,_8075,_8076,_8077,_8071,_8072,_8073,_8069,_8070,_8068]-
(clpfd:all_different([_8074,_8075,_8076,_8077,_8071,_8072,
_8073,_8069,_8070,_8068],[on(val)])),
(02) [_8074,_8075]-(clpfd:t=<u+c(_8074,_8075,-1)),
(03) [_8075,_8076]-(clpfd:t=<u+c(_8075,_8076,-1)),
(04) [_8076,_8077]-(clpfd:t=<u+c(_8076,_8077,-1)),
(05) [_8074,_8068]-(clpfd:t>=u+c(_8068,_8074,1)),
(06) [_8074,_8075,_8071]-(clpfd:x+y=t(_8071,_8074,_8075)),
(07) [_8074,_8076,_8072]-(clpfd:x+y=t(_8072,_8074,_8076)),
(08) [_8074,_8077,_8073]-(clpfd:x+y=t(_8073,_8074,_8077)),
(09) [_8075,_8076,_8069]-(clpfd:x+y=t(_8069,_8075,_8076)),
(10) [_8075,_8077,_8070]-(clpfd:x+y=t(_8070,_8075,_8077)),
(11) [_8076,_8077,_8068]-(clpfd:x+y=t(_8068,_8076,_8077)),
(12) [_8074]-(_8074 in 1..8),
(13) [_8075]-(_8075 in 3..12),
(14) [_8076]-(_8076 in 6..13),
(15) [_8077]-(_8077 in 10..15),
(16) [_8071]-(_8071 in 1..11),
(17) [_8072]-(_8072 in 3..12),
(18) [_8073]-(_8073 in 6..14),
(19) [_8069]-(_8069 in 1..10),
(20) [_8070]-(_8070 in 3..12),
(21) [_8068]-(_8068 in 2..9)]

————————————————————————————————————————
TOY(FDs)> golomb 5 true == M
————————————————————————————————————————
(01) [[_30421,_30422,_30423,_30424,_30425,_30426,_30427,_30428,
_30429,_30430]-(clpfd:all_different([_30421,_30422,_30423,
_30424,_30425,_30426,_30427,_30428,_30429,_30430],[on(val)])),
(02) [_30421,_30422]-(clpfd:t>=u+c(_30422,_30421,1)),
(03) [_30422,_30423]-(clpfd:t>=u+c(_30423,_30422,1)),
(04) [_30423,_30424]-(clpfd:t>=u+c(_30424,_30423,1)),
(05) [_30421,_30430]-(clpfd:t>=u+c(_30430,_30421,1)),
(06) [_30421,_30422,_30425]-(clpfd:x+y=t(_30421,_30425,_30422)),
(07) [_30421,_30423,_30426]-(clpfd:x+y=t(_30421,_30426,_30423)),
(08) [_30421,_30424,_30427]-(clpfd:x+y=t(_30421,_30427,_30424)),
(09) [_30422,_30423,_30428]-(clpfd:x+y=t(_30422,_30428,_30423)),
(10) [_30422,_30424,_30429]-(clpfd:x+y=t(_30422,_30429,_30424)),
(11) [_30423,_30424,_30430]-(clpfd:x+y=t(_30423,_30430,_30424)),
(12) [_30421]-(_30421 in 1..8),
(13) [_30422]-(_30422 in 3..12),
(14) [_30423]-(_30423 in 6..13),
(15) [_30424]-(_30424 in 10..15),
(16) [_30425]-(_30425 in 1..11),
(17) [_30426]-(_30426 in 3..12),
(18) [_30427]-(_30427 in 6..14),
(19) [_30428]-(_30428 in 1..10),
(20) [_30429]-(_30429 in 3..12),
(21) [_30430]-(_30430 in 2..9)]
```

Figure 8.4: Constraint Store in the SICStus and $\mathcal{TOY}(\mathcal{FD}s)$ Session

and $m_4 >= m_3 + 1$ (respectively).

Third, the lines (06)..(11) setting d_4,...,d_9 are also translated to indexicals following the pattern `x+y=t`, but the arguments `x` and `y` are inverted. Thus, the constraints of lines (06) and (11) are represented as $m_1 + d_4 = m_2$ and $m_3 + d_9 = m_4$ (respectively).

In summary, although SICStus and $\mathcal{TOY}(\mathcal{FD}s)$ follow the same formulation for G-5, the internal primitive constraint to indexical translation performed by `clpfd` differs in both models, leading to different filtering algorithms for some of the indexicals, and thus to a different performance of the search exploration.

8.6 Related Work

The literature contains a big amount of documents related to CP($\mathcal{FD}$) solving. As mentioned in Section 2.2, a main introduction to the constraint propagation and search exploration techniques any CP($\mathcal{FD}$) solver is based on can be found in Chapters 3 and 4 of [163], respectively. The former chapter presents the notions of arc and higher order consistencies, as well as variable domain inference by using constraint handling rules and specific constraints. The latter presents the notions of search tree, depth first search exploration, variable and value selection strategies, branch and bound optimization and restart strategies. Regarding the solving of CSP's and COP's for the algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$), CLP($\mathcal{FD}$) and CFLP($\mathcal{FD}$) systems considered in this chapter, once again, perhaps the best reference are the MiniZinc [133], ILOG OPL [108], Gecode [173], ILOG Solver [109], SICStus Prolog [126], SWI-Prolog [150], PAKCS [93] and $\mathcal{TOY}(\mathcal{FD})$ [40] user manuals, which include several case studies with a detailed explanation of the best configuration of the solver for tackling the problems.

Regarding comparison papers among different CP($\mathcal{FD}$) systems (cf. Section 7.8), [71] presents one based on eight C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$) systems: ECLiPSe, Oz, ILOG Solver (version 3.1), clp(fd) [58], CHR [75], SICStus `clpfd` (version 3.5), IF/Prolog [9] and B-Prolog. It compares a benchmark based on puzzle problems (with just some of them being scalable). For running the experiments, two different machines are used. Regarding the problem formulation, `all_different` global constraints are included if available, and both first-fail and naive labeling strategies are used. Moreover, the importance of a good specification is pointed out, by showing the improvement each system achieves when reformulating a problem using less variables and constraints. Regarding the solving results, ILOG Solver turns to be the fastest system. But, as a difference with the results of this chapter, it outperformed SICStus `clpfd` by 5-10 times. Thus, it can be seen that the performance gap between these two constraint libraries has been reduced w.r.t. the one they had 15 years ago. The comparison results also revealed ILOG Solver to have a better scalability than SICStus, allowing to solve larger problems (involving a greater number of variables). This gap has also been observed in $\mathcal{TOY}(\mathcal{FD})$, with $\mathcal{TOY}(\mathcal{FD}i)$ being able to solve larger instances than $\mathcal{TOY}(\mathcal{FD}s)$ for

the Magic Sequence and Queens problems (presented on Section 3.4).

The benchmark proposal of [71] was criticized in [197]. They claimed that some pitfalls were addressed, as providing a different specification for a same problem in different systems, use low-level implementation issues in some systems but not in others, and not using different machines for running the experiments of different systems. Moreover, they criticized using as benchmark a set of puzzle problems (as they are not representative of the issues found in real-life applications), and the lack of available source code for them. Unfortunately, some of these aspects are also applicable to the comparison of this chapter, as Section 8.1 has shown that it has been impossible to set-up a purely common framework for running the experiments (e.g., consistency-filtering algorithms for global constraints and CPU measurement for algebraic CP($\mathcal{FD}$) systems).

Regarding the comparison papers among CP($\mathcal{FD}$) and other techniques (cf. Section 7.8), [99] compares two different ILOG OPL models for an industrial planning problem, using ILOG Solver and ILOG CPLEX [13] as the target solvers, respectively. It concludes that the integer programming approach is more promising and reliable than the CP($\mathcal{FD}$) one, as it is faster for each instance being run. In particular, it points out that the linear relaxation improves the branch and bound search. Similarly, [53] compares two different ILOG OPL models for an Generic Supply Chain Model, but in this case the ILOG CPLEX ones uses Mixed Integer Programming. It can be seen that, whereas CP($\mathcal{FD}$) behaves quite well for small instances, for larger ones it does not find a solution after hours of computation (whereas the MIP approach finds an optimal solution in just 30 seconds).

Finally, [66] performs a comparison of classical and real-life problems in both CLP($\mathcal{FD}$) (using SICStus Prolog and B-Prolog) and ASP (using the answer set solvers Smodels and Cmodels, the latter relying in the SAT Solver mChaff). As general conclusions, they observe that graph-based problems have nice and compact encodings in ASP, and the performance of the ASP solutions is acceptable and scalable. On the other hand, for problems requiring more intense use of arithmetic and/or numbers, CLP($\mathcal{FD}$) efficiently handles them. More specifically, in most of the cases, both SICStus `clpfd` and Cmodels outperform Smodels (for Cmodels, the improvement can be directly assigned to the better performance of its underlying SAT solver). Regarding the comparison between SICStus and Cmodels, the system performing better is clearly dependent of the problem being run. Besides that, the paper also includes a brief comparison among the CLP($\mathcal{FD}$) systems SICStus Prolog, B-Prolog, GNU-Prolog and ECLiPSe, obtaining the same performance conclusions as in [71]. Moreover, similarly to the comparison performed in this chapter between each $\mathcal{TOY}(\mathcal{FD})$ version and its underlying solver, they provide a brief comparison between Cmodels and mChaff. The conclusions are, at some point, similar to the ones obtained in this chapter. That is, whereas ASP requires less encoding effort than modeling straight in the SAT solver, the former has a slight performance overhead w.r.t. the latter. However, they discover that, for a class of problems involving loop formulae, the ASP provides a better formulation

and also a better performance.

In summary, after considering related work, it can be said that this chapter provides a relevant contribution for comparing the solving performance of state-of-the-art CP($\mathcal{FD}$) systems, measuring the CPU time of each system for running the benchmark, and providing a low-level monitoring of each $\mathcal{TOY}(\mathcal{FD})$ version and its native constraint solver executions.

Besides these papers available in the literature, in the last years the MiniZinc Challenge [7] has emerged as a very suitable framework for comparing the performance of different constraint solvers. It has been celebrated each year since 2008, and provides an open forum in which the different participants can propose the problems of the competition. Then, there is a period of several weeks for tackling the problems and submitting solutions to the proposed instances. Whereas Gecode has won the competition in the period 2008-2012, in the last year the podium has been composed by Opturion/CPX [14], OR-Tools [15] and Gecode, respectively. Finally, as a side comment, a First International Lightning Model and Solve Competition [188] (celebrated in the last CP'13 conference) should be pointed out. On it, the conditions were quite different to the ones of the MiniZinc Challenge: Just 2 hours competition to tackle 6 new CSP's and COP's (with 3-5 instances each), teams of three people (but just using one laptop, which may contain as many different CP solvers as desired), and multiple points by submitting multiple suboptimal solutions to a very same instance. Under these circumstances, a team solving the instances by hand won the competition (defeating other 9 teams using CP solvers to model and solve the problem instances). This points out, once again, the complexity of tackling CSP's and COP's, and how even relying on expressive and efficient CP solvers, their adaptation to a concrete solver input takes time.

8.7 Conclusions

A common framework has been set for performing the experiments, considering the machine and system versions used, the set of variables and constraints (including the global ones), the propagation mode, the search strategy and the measurement of the CPU time and other search statistics. Any slight deviation of each system to this common setup has been justified.

G-9, G-10, G-11 and ETP-21 are classified as *js* as, for them, the three $\mathcal{TOY}(\mathcal{FD})$ versions spend above the 98% of the total CPU time (cf. tables 3.1 and 5.2), turning the performance comparison into purely solver dependent. ETP-7 and ETP-15 are classified as different factors *df* as, for them, the three $\mathcal{TOY}(\mathcal{FD})$ versions spend a percentage ranging in a 0%-98% of the total CPU time, turning the performance comparison into both CP($\mathcal{FD}$) and paradigm inherent overheads dependent.

The general performance results reveal that, for *js* instances, there is a clear performance ranking among the constraint solvers, with the positions 1-3, 4-6, 7-9 and

10 got by the Gecode, ILOG Solver, SICStus related systems and SWI, respectively. For *df* instances, this ranking among the constraint solvers is partially broken. On the one hand, SICStus and ILOG Solver obtain a better ranking for ETP-7 and ETP-15, and PAKCS, $\mathcal{TOY}(\mathcal{FD}s)$ and SWI-Prolog do it as well, but just for ETP-7. On the other hand, both MiniZinc and $\mathcal{TOY}(\mathcal{FD}i)$ obtain a worse ranking for both ETP-7 and ETP-15, and $\mathcal{TOY}(\mathcal{FD}g)$ and ILOG OPL do it as well, but just for ETP-7.

The complete ranking of the Golomb problem is homogeneous for the three instances: (1) MiniZinc; (2) Gecode; (3) $\mathcal{TOY}(\mathcal{FD}g)$; (4) $\mathcal{TOY}(\mathcal{FD}i)$; (5) ILOG Solver; (6) ILOG OPL; (7) SICStus; (8) PAKCS; (9) $\mathcal{TOY}(\mathcal{FD}s)$; and (10) SWI. Moreover, the slow-down of any system (w.r.t. the optimal MiniZinc) scales as the instances scale up (being around 2.0-2.5, 4.0-5.5 and 7.0-9.5 for the Gecode, ILOG Solver and SICStus related systems, respectively). However, if the best and worse Gecode, ILOG Solver and SICStus `clpfd` related systems are compared among them, then the slow-down of ILOG Solver and SICStus `clpfd` systems w.r.t. Gecode ones decreases in about a 50%-60%.

ETP is claimed to be a less homogeneous problem, as the complete ranking varies for the different instances tried (but, at least, Gecode ranks 1 for all of them). In ETP-21, the only *js* instance, the complete ranking is: (1) Gecode; (2) $\mathcal{TOY}(\mathcal{FD}g)$; (3) MiniZinc; (4) ILOG Solver; (5) $\mathcal{TOY}(\mathcal{FD}i)$; (6) ILOG OPL; (7) SICStus; (8) PAKCS; (9) $\mathcal{TOY}(\mathcal{FD}s)$; and (10) SWI-Prolog (with an slow-down being around 1.0-1.5, 1.9-2.5 and 3.8-6.8 for the Gecode, ILOG Solver and SICStus related systems, respectively).

Focusing on the three ETP instances, the slow-down between the systems ranking 2,...,9 w.r.t. Gecode decreases as the instances scale up. However, as the ranking is not stable, a head-to-head comparison between the slow-down of each concrete system w.r.t. Gecode reveals that there is not a general behavior, as depending on the system this slow-down can either decrease, increase, or even decrease and then increase as the instances scale up. Finally, if the best and worse Gecode, ILOG Solver and SICStus `clpfd` related systems are compared among them, then the ranking becomes nearly stable. However, the paradigm related overheads play a key role in ETP-7, modifying the performance order to (1) Gecode, (2) SICStus and (3) ILOG Solver, also decreasing the slow-down in a 40%-80% when the systems ranking 2 and 3 (respectively) are compared. In ETP-15, the overheads do not modify the performance order (1) Gecode, (2) ILOG Solver and (3) SICStus, but there is not a general pattern about how the slow-down behaves as the systems ranking 2 and 3 (respectively) are compared.

The Gecode, ILOG Solver and SICStus `clpfd` related systems have been respectively considered isolatedly. The instances G-11 and ETP-21 have been used to compare the results of $\mathcal{TOY}(\mathcal{FD}g)$ and Gecode; $\mathcal{TOY}(\mathcal{FD}i)$ and ILOG Solver; $\mathcal{TOY}(\mathcal{FD}s)$ and SICStus. The Gecode, ILOG Solver and SICStus `clpfd` constraint networks being built up (for these models) by the $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}s)$ interfaces have been monitored, as well as the search exploration being performed by them. These constraints networks and search exploration have been compared with the ones of the Gecode, ILOG Solver and SICStus models, monitoring any mismatch arisen between

each $\mathcal{TOY}(\mathcal{FD})$ execution and the one of its constraint solver native implementation.

The main conclusion for $\mathcal{TOY}(\mathcal{FD}g)$ is that its use is highly encouraged for solving these COP's, as it has been shown that the interface from $\mathcal{TOY}(\mathcal{FD})$ to the Gecode API builds up a Gecode model nearly equal to the C++ CP($\mathcal{FD}$) one (thus achieving the same performance as when programming in C++ CP($\mathcal{FD}$) directly using the Gecode API). In this setting, as both Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ contain nearly the same constraint network, the time they devote to search exploration nearly matches (there is a 1%-2% overhead for $\mathcal{TOY}(\mathcal{FD}g)$, due to the triggering of a few more constraints during the search exploration). For *df* instances, the performance of $\mathcal{TOY}(\mathcal{FD}g)$ is still worse than the one of Gecode, as it is penalized by the lazy narrowing and external solver overheads. But, for *js* instances, where this search exploration is the only factor determining the CPU time, $\mathcal{TOY}(\mathcal{FD}g)$ directly matches the CPU time of Gecode (1%-2% overhead), becoming nearly the second fastest system for solving pure CP($\mathcal{FD}$) problems as Golomb, and nearly the fastest system for solving real-life applications as ETP.

Also, the main conclusion for $\mathcal{TOY}(\mathcal{FD}i)$ is that its use is encouraged for solving these COP's, although its performance is clearly dependent on the formulation of the problem. The interface from $\mathcal{TOY}(\mathcal{FD})$ requires extra tuning to manage the ILOG Concert and ILOG Solver objects, which implies a relevant additional overhead. Whereas this overhead is crucial for *df* instances (with a quite big slow-down w.r.t. the execution of the native ILOG Solver model) it becomes nearly negligible for *js* ones. Moreover, this extra tuning produces a mismatch between the constraint network built up by $\mathcal{TOY}(\mathcal{FD}i)$ and the one built up by the C++ CP($\mathcal{FD}$) model (both in the amount of variables and constraints being used). It has been shwon that, whereas there are some situations in which $\mathcal{TOY}(\mathcal{FD}i)$ saves variables and constraints w.r.t. ILOG Solver, in other situations it is the other way round. Thus, the final constraint network applied in $\mathcal{TOY}(\mathcal{FD}i)$ and ILOG Solver is totally dependent on the formulation of the problem (more specifically, on the amount of these mismatch situations a formulation leads to). In this setting, $\mathcal{TOY}(\mathcal{FD}i)$ performs 6%-11% better than ILOG Solver for the Golomb instances (being nearly the fourth fastest system), and a 17% worse for ETP-21 (being nearly the fifth fastest system).

Finally, the main conclusion for $\mathcal{TOY}(\mathcal{FD}s)$ is that its use is encouraged for solving the pure CP($\mathcal{FD}$) COP of Golomb, but discouraged for solving the real-life COP of ETP. Regarding the former, the performance of $\mathcal{TOY}(\mathcal{FD}s)$ suffers an overhead ranging in a 7%-10% w.r.t. the CPU time when programming directly in SICStus by using the `clpfd` API. Besides that, $\mathcal{TOY}(\mathcal{FD}s)$ reaches nearly the same performance as when programming in the mate CFLP($\mathcal{FD}$) system PAKCS. However, when solving a real-life problem, the overhead of $\mathcal{TOY}(\mathcal{FD}s)$ ranges on a 70%-80% w.r.t. the CPU time of SICStus, being less competitive w.r.t. it and PAKCS. It has been shown that this performance mismatch is due to the constraint to indexical translation performed by `clpfd`, that differs in SICStus and $\mathcal{TOY}(\mathcal{FD}s)$ executions, giving rise to different constraint networks, i.e., different sets of indexicals (with different consistency-level propagation algorithms).

Part V

Conclusions and Perspectives

This part concludes the thesis. Chapter 9 gathers the main conclusions of the three parts of the research being accomplished, also presenting some perspectives for each of them.

Chapter 9

Conclusions and Perspectives

This chapter presents the main conclusions of the three research parts being accomplished in this thesis, respectively devoted to: The performance improvement of $\mathcal{TOY}(\mathcal{FD})$, real-life applications of $\mathcal{TOY}(\mathcal{FD})$, and a modeling and solving comparison of $\mathcal{TOY}(\mathcal{FD})$ w.r.t. state-of-the-art CP($\mathcal{FD}$) systems. Sections 9.1, 9.2 and 9.3 summarize the conclusions of each part, with Section 9.4 presenting some perspectives for them.

9.1 Improving the Performance of $\mathcal{TOY}(\mathcal{FD})$

A scheme for interfacing C++ CP($\mathcal{FD}$) solvers into $\mathcal{TOY}(\mathcal{FD})$ has been presented (easily adaptable to other CLP($\mathcal{FD}$) or CFLP($\mathcal{FD}$) Prolog systems). It has described the additional difficulties arisen in terms of communication with the solver and different variables, constraints and types representations. Also, it has described the adaptation of a C++ CP($\mathcal{FD}$) solver to the CFLP($\mathcal{FD}$) requirements of model reasoning, multiple search strategies (interleaved with constraint posting) and both incremental and batch propagation modes. The scheme has been shown to be generic enough, interfacing Gecode and ILOG Solver (leading to the new versions $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$, respectively) by finding no extra interface difficulties but the ones described in the scheme.

The performance of the three $\mathcal{TOY}(\mathcal{FD})$ versions has been measured by using three classical CP($\mathcal{FD}$) CSP's (Magic, Queens and Langford's) and a COP (Golomb). These problems cover the whole repertoire of $\mathcal{FD}$ constraints supported by $\mathcal{TOY}(\mathcal{FD})$. Also, by using three instances per problem (solved in the order of magnitude of tenths of seconds, seconds and minutes, respectively) the experiments have compared the $\mathcal{TOY}(\mathcal{FD})$ performance as the hardness of the problem scales.

It has been shown that $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ outperform $\mathcal{TOY}(\mathcal{FD}s)$, but the improvement achieved (ranging in 1.15-3.57 times faster) is dependent on the concrete problem and instance solved. For Magic and Golomb instances, the improvement

of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$ remains stable as the instances scale up. For Queens and Langford's instances, the improvement achieved scales from the instances solved in tenths of second to both the instances solved in seconds and minutes. In any case, the propagation mode does not play a key role in the solving time, as although batch mode is faster for all the instances, the differences achieved are very small (about an order of magnitude smaller than the CPU solving time of the instance, or even less).

Moreover, there is a clear correlation between the percentage of CPU solving time devoted to search exploration and the improvement achieved. Thus, whereas this percentage remains stable in Magic and Golomb, it clearly scales from the Queens and Langford's instances solved in tenths of second to both the instances solved in seconds and minutes. This turns the performance of $\mathcal{TOY}(\mathcal{FD})$ into purely CP($\mathcal{FD}$) dependent. That is, the CPU time of each version directly comes from the performance of its concrete solver in achieving the pure CP($\mathcal{FD}$) mechanism of performing a search exploration by propagating basic and global constraints.

Thus, another suitable approach to improve the solving performance of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ has been to focus on the search strategy, replacing it by an *ad hoc* one which exploits the knowledge about the structure of the problem and its solutions. That is, keeping the same solver to accomplish the search, modify the $\mathcal{TOY}(\mathcal{FD})$ model to specify a search strategy requiring less search exploration to find the solutions. The motivation of this approach (easily adaptable to other CLP($\mathcal{FD}$) or CFLP($\mathcal{FD}$) Prolog systems interfacing C++ CP($\mathcal{FD}$) solvers) has been to take advantage of both the high expressivity of $\mathcal{TOY}(\mathcal{FD})$ for specifying search strategies, and of the high efficiency of Gecode and ILOG Solver for accomplishing them.

Specifically, eight new search primitives have been described, which have included some novel search concepts (not directly available neither in Gecode nor in ILOG Solver), as performing an exhaustive breadth exploration of the search tree (further sorting the satisfiable solutions by a specified criterion), fragmenting the variables pruning each one to a subset of its domain values (instead of binding it to a single value), and applying the labeling or fragment strategy only to a subset of the variables involved. It has been pointed out how expressive, easy and flexible it is to specify some search criterion at $\mathcal{TOY}(\mathcal{FD})$ level, as well as how easy it is to use model reasoning to apply different search strategies (setting different search scenarios) to the solving of a problem.

The new versions of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ have extended the Gecode and ILOG Solver libraries by relying on their different underlying search mechanisms (`Search Engine`, `Brancher` methods and hybrid recomputation for Gecode; `IloGoal`, goal constructor and goal stack for ILOG Solver). An abstract view of the requirements needed to integrate the search strategies in $\mathcal{TOY}(\mathcal{FD})$ has been first presented (with the scheme further instantiated to Gecode and ILOG Solver). Also, the impact of the implementation of the search primitives on the system architecture has been analyzed, revealing that the specification of search criterion at $\mathcal{TOY}(\mathcal{FD})$ level

has an inherent overhead due to the recursive interaction between the SICStus Prolog and C++ layers.

The $\mathcal{TOY}(\mathcal{FD})$ models for Magic, Queens, Langford's and Golomb have been revisited, discussing the structure of the solutions of each problem, and pointing out how the use of the proposed search strategies allow to reduce the search exploration to find them. Mate $\mathcal{TOY}(\mathcal{FD})$ models with an improved *ad hoc* search strategy have been developed, showing that the solving time of both $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ is improved (in a range of 1.05 times faster to more than 1000 times) w.r.t. the solving time of original models. In particular, whereas for Queens and Langford's instances the better performance achieved clearly scales as the size of the instances scale, for Magic ones it remains stable, and for Golomb ones it decreases. Moreover, the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ is greater for the new improved $\mathcal{TOY}(\mathcal{FD})$ models, revealing that the approach Gecode offers to extend the library with new search strategies is more efficient than the one of ILOG Solver.

9.2 Real-Life Applications of $\mathcal{TOY}(\mathcal{FD})$

Two real-life applications of the system have been presented. The first one has been a Employee Timetabling Problem (ETP) coming from the communication industry. In contrast to the simple formulation of the classical CP($\mathcal{FD}$) problems (Magic Series, Queens, Langford's Number and Golomb Rulers) this ETP exploits both the high expressivity of $\mathcal{TOY}(\mathcal{FD})$ and its higher solving performance just achieved.

A solving approach to tackle the problem has been presented, fully parametric, non-monolithic and including CP($\mathcal{FD}$) independent components. Three instances (with solving times of tenths of seconds, seconds and minutes, respectively) have been proposed, to compare the conclusions for ETP with the ones for the classical CP($\mathcal{FD}$) problems. It has been shown that ETP behaves similarly to Queens and Langford's: The $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ improvement w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$ increases as the instances scale up, as it increases the CPU solving time devoted to search exploration and the impact of the improved search strategy applied.

However, the results for ETP are more dependent on the concrete instance being run than when solving Queens or Langford's: $\mathcal{TOY}(\mathcal{FD}g)$ still outperforms $\mathcal{TOY}(\mathcal{FD}s)$, but the range of its improvement achieved is wider than the one for Queens and Langford's. $\mathcal{TOY}(\mathcal{FD}i)$ behaves worse, equivalent or better than $\mathcal{TOY}(\mathcal{FD}s)$ for the three ETP instances, respectively. In any case, the search exploration percentage of $\mathcal{TOY}(\mathcal{FD}g)$ and $\mathcal{TOY}(\mathcal{FD}i)$ for the ETP instances solved in tenths of second and seconds is much smaller than the one of $\mathcal{TOY}(\mathcal{FD}s)$, revealing that more time is spent in the interface with the external CP($\mathcal{FD}$) solvers. Regarding propagation mode, batch clearly improves incremental in $\mathcal{TOY}(\mathcal{FD}i)$ for the ETP instances solved in tenths of second and seconds, and in $\mathcal{TOY}(\mathcal{FD}s)$ for the ETP instances solved in seconds and

minutes. Finally, the impact of applying the improved search strategy is smaller in ETP instances than in Queens and Langford's ones. Moreover, the speed-up of $\mathcal{TOY}(\mathcal{FD}g)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}i)$ does not increase as the instances scale up.

Second, an empirical analysis of the solving hardness of the Bin Packing Problem (BPP) has been performed. It has included the solving of a parametric generated BPP benchmark (based on the well known Weibull distribution), applying two equivalent CP($\mathcal{FD}$) models (Gecode and $\mathcal{TOY}(\mathcal{FD}g)$) and four heuristics (MAXREST, FIRSTFIT, BESTFIT and NEXTFIT). The conclusions are interesting for the future development of portfolio solvers, tackling configuration problems (which can be seen as generalizations of the BPP) coming from the industry of the data centre optimization.

The flexibility of Weibull has been discussed, showing that a great variety of item size distributions can be generated via different combinations of its (k,λ) parameters. A set of real-life BPP instances have been very accurately modeled using Weibull (MLF and Q-Q plots have visually shown the quality of the fit, and the K-S and χ^2 statistical tests have rigorously ensured it). A benchmark suite of 19,900 instances has been generated, consisting of 199 different categories (of 100 instances of 100 items each) or combinations of (k,λ). Specifically, λ has been fixed to 1000 (spanning the item sizes of the distributions over three orders of magnitude), and k has ranged in $k = [0.1, 0.2, \ldots, 19.9]$. Once the instance set is fixed, 11 different bin capacities C have been tried, setting it to the size of the highest item of the instance times a factor ranging from 1.0 to 2.0 (increasing it 0.1 on each new scenario).

The analysis of the CP($\mathcal{FD}$) results have revealed that, for C parameter, there are categories (k,λ) for which the CP($\mathcal{FD}$) approach finds hard to solve the whole instance set of 100 instances. Moreover, the hardness a concrete category results for the CP($\mathcal{FD}$) method can be classified by using 5 groups: 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%. Thus, it can be seen that increasing C also increases the computational challenge of the problem, so that the amount of instances being solved decrease as C increases. In $C = 1.0$, 178 categories belong to group 80%-100% and 21 to group 60%-80%. In $C = 1.5$, 150 categories belong to group 80%-100%, 17 to group 60%-80%, 18 to group 40%-60% and 14 to group 20%-40%. In $C = 2.0$, 87 categories belong to group 80%-100%, 20 to group 60%-80%, 12 to group 40%-60%, 18 to group 20%-40% and 62 to group 0%-20% (the latter with up to 12 categories for which the percentage of instances being solved reached nearly a 0%). Also, there is a correlation between the percentage of instances solved and the time devoted to search exploration.

Also, as k increases, the amount of bins used in the optimal solutions being found increases as well. As all the instances contain 100 items, there are concrete intervals of items per bins for the categories belonging to groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%. However, these intervals slightly change for the different C values. On the one hand, it can be concluded that, if the category requires an average of less that 1.60 or more than 4.00 items per bin, then the category is going to be classified into group 80%-100%. On the other hand, for the rest of groups there are

no general conclusions for the different C configurations. Specifically, for $C = 2.0$ and $k = [10.0, \ldots, 19.9]$ there are categories of groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20%, all of them with a same amount of 2.0 items per bin.

The heuristics solve the entire benchmark in a negligible amount of time, and thus the analysis has focused on the average deviations of MAXREST (with similar results to FIRSTFIT, and BESTFIT) and NEXTFIT w.r.t. the optimal number of bins found by CP($\mathcal{FD}$). The deviations of the former are an order of magnitude better (ranging in 0.0-1.4 bins) than the ones of the latter (ranging in 0.0-20.0 bins). In MAXREST, it is clear that, as C increases, the average gap achieved by the heuristic w.r.t. the CP($\mathcal{FD}$) optimal solutions increases as well. Moreover, there is a clear correlation between the categories belonging to groups 80%-100%, 60%-80%, 40%-60%, 20%-40% and 0%-20% and the gap achieved by the heuristic. In NEXTFIT, there is a frontier in $C = 1.5$. Thus, whereas from $C = 1.0$ to $C = 1.5$ the gap increases, from $C = 1.6$ to $C = 2.0$ it decreases. Unfortunately, there is no correlation between the categories and the gap achieved. In general, whereas the heuristic MAXREST (and thus also of FIRSTFIT and BESTFIT) represents a very good alternative to the CP($\mathcal{FD}$) approach (as it achieves very good solutions in a nearly negligible amount of time), the gap achieved by NEXTFIT makes it not that much interesting.

9.3 Positioning $\mathcal{TOY}(\mathcal{FD})$ w.r.t. Other CP($\mathcal{FD}$) Systems

The Golomb and ETP COP's have been used to perform a modeling and solving comparison among the state-of-the-art algebraic CP($\mathcal{FD}$) systems MiniZinc and ILOG OPL, the C++ CP($\mathcal{FD}$) systems Gecode and ILOG Solver, the CLP($\mathcal{FD}$) systems SICStus Prolog and SWI-Prolog, and the CFLP($\mathcal{FD}$) systems PAKCS and $\mathcal{TOY}(\mathcal{FD})$ (with its three different versions $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ and $\mathcal{TOY}(\mathcal{FD}s)$).

The main conclusion for $\mathcal{TOY}(\mathcal{FD})$ is that its use is an appealing alternative for modeling these COP's, because of a number of advantages: It abstracts the notion of the constraint solver, isolating the use of several solvers and the distribution of the constraints on them. It also supports free access to the variables. It allows to model the problems in just one file, matching the multiple-stage architecture of the p_tt algorithm by simply placing the stages in order. It uses dynamic data structures, and makes it easy to access an index them. It allows to save several $\mathcal{FD}$ variables, placing first the variable unifications and then the rest of the $\mathcal{FD}$ constraints. It provides batch and incremental primitives for easily applying different propagation modes to different parts of the program. It is a declarative general-purpose programming language, including expressive modeling features such as non-deterministic functions, types, higher-order functions, lazy evaluation, pattern matching and partial application, for allowing the user to write neater formulations. Thus, only the algebraic CP($\mathcal{FD}$) systems require less amount of lines of code for modeling the problems.

The main conclusion for $\mathcal{TOY}(\mathcal{FD}g)$ is that its use is highly encouraged for solving these COP's, as it has been shown that the interface from $\mathcal{TOY}(\mathcal{FD})$ to the Gecode API builds up a Gecode model nearly equal to the C++ CP($\mathcal{FD}$) one (thus achieving the same performance as when programming a Gecode native model). In this setting, as both Gecode and $\mathcal{TOY}(\mathcal{FD}g)$ contain nearly the same constraint network, the time they devote to search exploration nearly matches (there is a 1%-2% overhead for $\mathcal{TOY}(\mathcal{FD}g)$, due to the triggering of a few more constraints during the search exploration). For *df* instances, the performance of $\mathcal{TOY}(\mathcal{FD}g)$ is still worse than the one of Gecode, as it is penalized by the lazy narrowing and external solver overheads. But, for *js* instances, where this search exploration is the only factor determining the CPU time, $\mathcal{TOY}(\mathcal{FD}g)$ directly matches the CPU time of Gecode (1%-2% overhead), becoming nearly the second fastest system for solving Golomb, and nearly the fastest system for solving ETP.

The main conclusion for $\mathcal{TOY}(\mathcal{FD}i)$ is that its use is encouraged for solving these COP's, although its performance is clearly dependent on the formulation of the problem. The interface from $\mathcal{TOY}(\mathcal{FD})$ requires extra tuning to manage the ILOG Concert and ILOG Solver objects, which implies a relevant additional overhead. Whereas this overhead is crucial for *df* instances (with a quite big slow-down w.r.t. the execution of the native ILOG Solver model) it becomes nearly negligible for *js* ones. Moreover, this extra tuning produces a mismatch between the constraint network built up by $\mathcal{TOY}(\mathcal{FD}i)$ and the one built up by the native ILOG Solver model (both in the amount of variables and constraints being used). It has been shown that, whereas there are some situations in which $\mathcal{TOY}(\mathcal{FD}i)$ saves variables and constraints w.r.t. ILOG Solver, in other situations it is the other way round. Thus, the final constraint network applied in $\mathcal{TOY}(\mathcal{FD}i)$ and ILOG Solver is totally dependent on the formulation of the problem (more specifically, on the amount of these mismatch situations a formulation leads to). In this setting, $\mathcal{TOY}(\mathcal{FD}i)$ performs 6%-11% better than ILOG Solver for the Golomb instances (being nearly the fourth fastest system) and a 17% worse for ETP-21 (being nearly the fifth fastest system).

The main conclusion for $\mathcal{TOY}(\mathcal{FD}s)$ is that its use is encouraged for solving the pure CP($\mathcal{FD}$) COP of Golomb, but discouraged for solving the real-life COP of ETP. Regarding Golomb, the performance of $\mathcal{TOY}(\mathcal{FD}s)$ suffers an overhead ranging in a 7%-10% w.r.t. the native SICStus model. Besides that, $\mathcal{TOY}(\mathcal{FD}s)$ reaches nearly the same performance as PAKCS. However, regarding ETP, the overhead of $\mathcal{TOY}(\mathcal{FD}s)$ ranges on a 70%-80% w.r.t. the native SICStus model, being less competitive w.r.t. it (and even w.r.t. PAKCS). It has been shown that this performance mismatch is due to the constraint to indexical translation performed by `clpfd`, that differs in SICStus and $\mathcal{TOY}(\mathcal{FD}s)$ executions, giving rise to different constraint networks, i.e., different sets of indexicals (relying on different consistency-level propagation algorithms).

In summary, besides the general interest of comparing these state-of-the-art CP($\mathcal{FD}$) systems, the results have shown $\mathcal{TOY}(\mathcal{FD})$ to be competitive w.r.t. any of them for both the modeling and solving of these COP's, thus encouraging its use (and the use of

the CFLP($\mathcal{FD}$) paradigm itself).

9.4 Future Work

The generic scheme proposed in Chapter 3 can be useful for interfacing new C++ CP($\mathcal{FD}$) solvers (and providing a better performance than the ones of Gecode and ILOG Solver). Besides that, the complex nature of many CSP's and COP's makes their specification to contain several variable domains. Although Gecode and ILOG Solver support integer, Boolean, set and even float variables, sometimes the computational challenge of these CSP's and COP's just make unrealistic to solve the problem by simply applying an exhaustive search exploration. A clear example was shown with the generalized BPP's of the data centre optimization industry, where hard and dynamic planning problems were proposed. In these cases, the combination of CP($\mathcal{FD}$) with other techniques is required. For example, the Lazy Clause Generation [187] has recently combined CP($\mathcal{FD}$) with SAT solving, and it has been shown successful for solving many well-known real-life open problems [174].

In $\mathcal{TOY}$, there is a system version combining CP($\mathcal{FD}$) with Mathematical Programming techniques [68] via the coordination of the SICStus `clpfd` and `clpr` constraint libraries (from now on $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$). A specific mediatorial solver $\mathcal{M}$ manages the bridge constraints X `#==` Y arisen in the goal computation. Each of these constraints requests the float variable Y to take an integral real value equivalent to that of the integer variable X. Then, the `clpfd` and `clpr` solvers work independently, but explicit bridge projections [104] make any pruning in X or Y to be automatically shared with the mate solver and variable. Thus, the system $\mathcal{TOY}(\mathcal{FD}i)$ can be extended to reproduce the proposed scheme, with ILOG Solver playing the role of SICStus `clpfd` and ILOG CPLEX [13] playing the role of SICStus `clpr`. The better performance of ILOG Solver and ILOG CPLEX w.r.t. SICStus `clpfd` and `clpr` (respectively) will improve the solving performance of the new system w.r.t. the existing version (so as the better performance of ILOG Solver w.r.t. SICStus `clpfd` improved the solving performance of $\mathcal{TOY}(\mathcal{FD}i)$ w.r.t. $\mathcal{TOY}(\mathcal{FD}s)$). Moreover, both ILOG Solver and ILOG CPLEX make use of the same ILOG Concert modeling library. The new system version could implement the mediatorial solver $\mathcal{M}$ in ILOG Concert, leveraging it from the Prolog layer of the architecture of the system, thus improving even more the solving performance. As a drawback, it has to be said that ILOG CPLEX always computes an extensional solution of the proposed equation system (binding variables to values), instead of an intensional solution (simplifying the system), as SICStus `clpr` does.

The *ad hoc* search strategies proposed in Chapter 4 could be applied to classical CP($\mathcal{FD}$) benchmarks under multiple and very precisely controlled scenarios, by using scripting techniques as the ones proposed for $\mathcal{TOY}(\mathcal{FD}g)$ in Chapter 6 (based in non-deterministic functions). In them, an exhaustive combination of applying one or dif-

ferent search strategies (as well as the variable subset used on each of them) can be analyzed. Then, the results could be analyzed, in order to find out which strategies had lead to a solution, or to a minimum search space containing a solution. Moreover, this analysis will help to find out new patterns about the relation between the structure of a concrete problem and the concrete search strategy (or combination of search strategies) to be applied to successfully solve it.

The ETP model of Chapter 5 is fully parametric, so multiple new instances should be tried, ranging in the number of days of the timetable, number of teams, workers per team, periodicity the extra worker can be selected, the extra factor its working hours must be paid, number of different kinds of working days, absences of the regular workers of the teams and how tight the shifts must be distributed among the workers of a team. Moreover, the use of multiple disjoint teams is a particular characteristic of the formulation, but the possibility of using a single team can also be considered (by simply setting that concrete parameter to value one), thus increasing the applicability of the proposed model to many other timetabling real-life problems.

The conclusions of the empirical BPP analysis of Chapter 6 represent a basis for the future development of portfolio solvers, tackling generalized BPP real-life instances coming from the data centre optimization industry. By applying Maximum Likelihood Fitting to the real-life instance, it is easy to find the Weibull parameters (k,λ) best fitting it. Then, by looking back to the concrete results of that particular Weibull parameter combination category, it would be straightforward to determine if it is better to apply a CP($\mathcal{FD}$) or a heuristic approach to tackle it. Finally, the approach of generating a new benchmark (as a way to study real-life instances) can be extended to other important problems, such as knapsacks, multi-processor scheduling or job shop scheduling, to name but a few.

Also, as already mentioned, many real-life applications can be found for CP($\mathcal{FD}$). Just focusing on the papers published in the application track of the last CP'13 edition, a wide range of applications domains can be found, including: Power Stations Planning [26], Agricultural Land Allocation [35], Bike Sharing Systems [77], Chaotic Dynamic Systems [84], Laser Cutting Path Planning [117], Atom Mapping [129], Hardware Verification [140], Wireless Network Assignment [143], Multiple-Choice Math Quizzes [191], Wine Blending [196] and Berth Allocation & Quay Crane Assignment [208]. Thus, it could be discussed the application of $\mathcal{TOY}(\mathcal{FD})$ to some of these (and other related) problems and application domains.

Regarding comparing CP($\mathcal{FD}$) systems, as in Chapters 7 and 8, obviously more classical CP($\mathcal{FD}$) problems (as the ones of CSPlib) and real-life problems (as the ones just said before) can be considered. In the case of the ETP, and following the proposal of evaluating multiple new instances exploiting the parametric the problem is, these new set of ETP instances can be also applied to the solving comparison. Also, more systems can be considered in the algebraic CP($\mathcal{FD}$), C++ CP($\mathcal{FD}$) and CLP($\mathcal{FD}$) paradigms, as the ones mentioned for each paradigm in Chapter 2.

Besides that, $\mathcal{TOY}(\mathcal{FD})$ could be positioned w.r.t. the application of other techniques different to CP($\mathcal{FD}$), as Mathematical Programming and SAT solving. First, the ETP problem can be modeled with ILOG CPLEX, developing a native C++ CP($\mathcal{R}$) ILOG CPLEX model. A modeling comparison of the ILOG Solver, $\mathcal{TOY}(\mathcal{FD}i)$ and the ILOG CPLEX models will allow to discuss the advantages that CP($\mathcal{FD}$) provides for formulating such a constraint-oriented problem. Also, a solving comparison will allow to compare the performance of applying an exhaustive search or a simplex approach [141] to the solving of multiple ETP instances. Second, the new $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$ version proposed can be also applied to the problem, relying on the support of mathematical techniques by interfacing ILOG CPLEX. Thus, a modeling comparison between the new $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$ model and the native C++ CP($\mathcal{R}$) ILOG CPLEX one will allow to evaluate how the higher abstraction of $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$ eases the formulation of the problem, and to compare this abstraction with the one achieved for ILOG Solver and $\mathcal{TOY}(\mathcal{FD})$. Also, a comparison between the performance of $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$ and ILOG CPLEX will allow to analyze if the interface from $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$ to ILOG CPLEX produces the same kind of mismatches in the ILOG CPLEX constraint solver input as the ones arisen between the $\mathcal{TOY}(\mathcal{FD})$ and the ILOG Solver native models (the constraint network posted to the solver was dependent on the formulation of the problem). Finally, a possible reformulation of the ETP problem could be discussed, in order to suit it to a SAT solver.

Bibliography

[1] 2012 ROADEF/EURO Challenge. http://challenge.roadef.org/2012/en/index.php.

[2] Bin Packing Problem. http://en.wikipedia.org/wiki/Bin_packing_problem.

[3] Chi-squared test. http://mathworld.wolfram.com/Chi-SquaredTest.html.

[4] Ciao Prolog. http://clip.dia.fi.upm.es/Software/Ciao/.

[5] GNU Prolog. http://www.gprolog.org/.

[6] Kolmogorov-smirnov test. http://mathworld.wolfram.com/Kolmogorov-SmirnovTest.html.

[7] MiniZinc Challenge. http://www.minizinc.org/challenge2013/challenge.html.

[8] The R Project for Statistical Computing. Version 1.47.0. http://www.r-project.org/.

[9] IF/Prolog V5.0, 1994.

[10] Alice ML, 2007. www.ps.uni-saarland.de/alice.

[11] Cmodels, 2010. http://www.cs.utexas.edu/users/tag/cmodels/.

[12] IBM ILOG CP 1.6, 2010. http://www-947.ibm.com/support/entry/portal/Overview/Software/WebSphere/IBM_ILOG_CP.

[13] IBM ILOG CPLEX 12.2, 2013. http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r2/index.jsp.

[14] Opturion CPX: Discrete Optimizer, 2013. http://www.opturion.com/.

[15] OR-Tools: Operation Research Tools Developed at Google, 2013. https://code.google.com/p/or-tools/.

[16] The ECLiPSe Constraint Programming System, 2013. http://eclipseclp.org/.

[17] A. C. F. Alvim, C. C. Ribeiro, F. Glover, and D. J. Aloise. A Hybrid Improvement Heuristic for the One-Dimensional Bin Packing Problem. *Journal of Heuristics*, 10(2):205–229, 2004.

[18] S. Antoy. Definitional Trees. In *Algebraic and Logic Programming, 143–157*. Springer, 1992.

[19] S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, 53, 2010.

[20] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[21] R. Bai, E. K. Burke, G. Kendall, J. Li, and B. McCollum. A Hybrid Evolutionary Approach to the Nurse Rostering Problem. *Trans. Evol. Comp*, 14(4):580–590, 2010.

[22] S. Bardin and A. Gotlieb. FDCC: A Combined Approach for Solving Constraints over Finite Domains and Arrays. In *CPAIOR'12, 17–33*. LNCS 7298. Springer, 2012.

[23] R. Barták. Constraint Propagation and Backtracking-Based Search. `http://ktiml.mff.cuni.cz/~bartak/downloads/CPschool05notes.pdf`.

[24] N. Beldiceanu, M. Carlsson, P. Flener, and J. Pearson. On Matrices, Automata, and Double Counting in Constraint Programming. *Constraints*, 18(1):108–140, 2013.

[25] N. Beldiceanu, M. Carlsson, T. Petit, and J.-C. Régin. An O(nlog n) Bound Consistency Algorithm for the Conjunction of an alldifferent and an Inequality between a Sum of Variables and a Constant, and its Generalization. In *ECAI'12, 145–150*. IOS Press, 2012.

[26] N. Beldiceanu, G. Ifrim, A. Lenoir, and H. Simonis. Describing and Generating Solutions for the EDF Unit Commitment Problem with the ModelSeeker. In *CP'13, 733–748*. LNCS 8124, Springer, 2013.

[27] C. Bessiere. Arc-Consistency and Arc-Consistency Again. *Artificial Intelligence*, 65:179–190, 1994.

[28] C. Bessiere, E. Freuder, and J. Régin. Using Constraint Metaknowledge to Reduce Arc Consistency Computation. *Artificial Intelligence*, 107:125–148, 1999.

[29] C. Bessiere and J. Régin. Refining the Basic Constraint Propagation Algorithm. In *IJCAI'01, 309–315*. Morgan Kaufmann, 2001.

[30] Boost: Free Peer-Reviewed Portable C++ Source Libraries. Version 1.47.0. `http://www.boost.org/`.

[31] Bprolog. `http://www.probp.com/`.

[32] N. Brauner, R. Echahed, G. Finke, H. Gregor, and F. Prost. Specializing Narrowing for Timetable Generation: A case study. In *PADL'05, 22–36*. LNCS 3350. Springer, 2005.

[33] K. B. Bruce. *Foundations of object-oriented languages - types and semantics*. MIT Press, 2002.

[34] T. Budiono and K. Wong. A Pure Graph Coloring Constructive Heuristic in Timetabling. In *ICCIS'12, 365–371*. IEEEXplore, 2012.

[35] Q. Bui, Q. Pham, and Y. Deville. Solving the Agricultural Land Allocation Problem by Constraint-Based Local Search. In *CP'13, 749–757*. LNCS 8124, Springer, 2013.

[36] D. Bulka and D. Mayhew. *Efficient C++: Performance Programming Techniques*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[37] E. Burke, G. Kendall, M. Mısır, and E. Özcan. Monte Carlo Hyper-heuristics for Examination Timetabling. *Annals of Operations Research*, 196(1), 2012.

[38] E. K. Burke, P. D. Causmaecker, G. V. Berghe, and H. V. Landeghem. The State of the Art of Nurse Rostering. *J. Scheduling*, 7(6):441–499, 2004.

[39] E. K. Burke, M. Gendreau, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and R. Qu. Hyper-heuristics. *Journal of the Operational Research Society*, 64(12):1695–1724, 2013.

[40] R. Caballero and J. Sánchez. TOY, A Multiparadigm Declarative Language, 2011. `http://sourceforge.net/projects/toy/files/toy/toy2.3.2/toyreport2.3.2.v1.0.pdf/download`.

[41] Cacao. `http://cacao.sourceforge.net/`.

[42] H. Cambazard and B. O'Sullivan. Propagating the Bin Packing Constraint Using Linear Programming. In *CP'10, 129–136*. LNCS 6308. Springer, 2010.

[43] M. Carlsson, N. Beldiceanu, and J. Martin. A Geometric Constraint over k-Dimensional Objects and Shapes Subject to Business Rules. In *CP'08, 220–234*. LNCS 5202. Springer, 2008.

[44] M. Carlsson and P. Mildner. SICStus Prolog - The first 25 years. *TPLP*, 12(1-2):35–66, 2012.

[45] I. Castiñeiras, M. De Cauwer, and B. O'Sullivan. Weibull-based Benchmarks for Bin Packing. In *CP'12, 207–222*. LNCS 7514. Springer, 2012.

[46] I. Castiñeiras and F. Sáenz-Pérez. Integration of ILOG CP into TOY. In *WFLP'09, 27–43*. LNCS 5979, 2009.

[47] I. Castiñeiras and F. Sáenz-Pérez. A CFLP Approach for Modeling and Solving a Real Life Employee Timetabling Problem. In *COPLAS'11, 63–70*, 2011. http://icaps11.icaps-conference.org/proceedings/coplas/coplas2011-proceedings.pdf.

[48] I. Castiñeiras and F. Sáenz-Pérez. Improving the Performance of FD Constraint Solving in a CFLP System. In *FLOPS'12, 88–103*. LNCS 7294. Springer, 2012.

[49] I. Castiñeiras and F. Sáenz-Pérez. Applying CP(FD), CLP(FD) and CFLP(FD) to a Real-Life Employee Timetabling Problem. *Procedia Computer Science*, 18(0):531 – 540, 2013.

[50] I. Castiñeiras and F. Sáenz-Pérez. Comparing TOY(FD) with State-of-the-Art Constraint Programming Systems. Technical report, 2013. http://federwin.sip.ucm.es/sic/investigacion/publicaciones/informes-tecnicos.

[51] I. Castiñeiras and F. Sáenz-Pérez. Improving the Search Capabilities of a CFLP System. In *PROLE'13, 273–287*. Under consideration for publication in Electronic Communications of the EASST, 2013. http://www.congresocedi.es/images/site/actas/ActasPROLE_TPF.pdf.

[52] P. H. Cheong and L. Fribourg. Implementation of Narrowing: The Prolog-Based Approach. In *Logic Programming Languages: Constraints, Functions, and Objects*, pages 1–20. MIT Press, 1993.

[53] D. Chippington, C. Lucas, and G. Mitra. An Investigation Comparing Mixed Integer Programming and Constraint Programming on a Generic Supply Chain Model, 2000. http://www.carisma.brunel.ac.uk/presentations/CARISMAApmod.pdf.

[54] Choco: an open source Java constraint programming library. http://sourceforge.net/projects/choco/.

[55] S. Chrétien, J.-M. Nicod, L. Philippe, V. Rehn-Sonigo, and L. Toch. Job Scheduling Using Successive Linear Programming Approximations of a Sparse Model. In *Euro-Par'12, 116–127*. LNCS 7484. Springer, 2012.

[56] G. Chu, M. Banda, and P. Stuckey. Exploiting Subproblem Dominance in Constraint Programming. *Constraints*, 17(1):1–38, 2012.

[57] G. Chu and P. J. Stuckey. Chuffed: A lazy clause solver, 2010. http://www.minizinc.org/challenge2010/description_chuffed.txt.

[58] P. Codognet and D. Diaz. Compiling Constraints in clp(fd). *The Journal of Logic Programming*, 19&20:503–581, 1994.

[59] S. Colton. The Painting Fool: Stories from Building an Automated Painter. In *Computers and Creativity*. Springer, 2012.

[60] Comet 2.1. http://dynadec.com/technology/constraint-programming/.

[61] J. V. de Carvalho. Exact Solution of Cutting Stock Problems Using Column Generation and Branch-and-Bound. *International Transactions in Operational Research*, 5(1):35–44, 1998.

[62] M. de la Banda, D. Jeffery, K. Marriott, N. Nethercote, P. Stuckey, and C. Holzbaur. Building Constraint Solvers with HAL. In *ICLP'01, 90–104*. LNCS 2237. Springer, 2001.

[63] R. Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[64] M. L. Delignette-Muller, R. Pouillot, J. B. Denis, and C. Dutang. *Use of the Package fitdistrplus to Specify a Distribution from Non-censored or Censored Data*, 2011.

[65] A. K. Dewdney. Computer Recreations. *Scientific American*, pages 14–21, 1986.

[66] A. Dovier, A. Formisano, and E. Pontelli. An Empirical Study of Constraint Logic Programming and Answer Set Programming Solutions of Combinatorial Problems. *J. Exp. Theor. Artif. Intell.*, 21(2):79–121, 2009.

[67] J. Dupuis, P. Schaus, and Y. Deville. Consistency Check for the Bin Packing Constraint Revisited. In *CPAIOR'10, 117–122*. LNCS 6140. Springer, 2010.

[68] S. Estévez-Martín, A. Fernández, M. Hortalá-González, F. Sáenz-Pérez, M. Rodríguez-Artalejo, and R. del Vado-Vírseda. On the Cooperation of the Constraint Domains H, R and FD in $CFLP$. *TPLP*, 9(4): pages 415–527, 2009.

[69] FaCiLe. http://www.recherche.enac.fr/opti/facile/.

[70] E. Falkenauer. A Hybrid Grouping Genetic Algorithm for Bin Packing. *Journal of Heuristics*, 2(1):5–30, 1996.

[71] A. J. Fernández and P. M. Hill. A Comparative Study of Eight Constraint Programming Languages Over the Boolean and Finite Domains. *Constraints*, 5(3):279–305, 2000.

[72] A. J. Fernández, T. Hortalá-González, F. Sáenz-Pérez, and R. del Vado-Vírseda. Constraint Functional Logic Programming over Finite Domains. *TPLP*, 7(5):537 – 582, 2007.

[73] K. Francis, S. Brand, and P. Stuckey. Optimisation Modelling for Software Developers. In *CP'12, 274–289*. LNCS 7514. Springer, 2012.

[74] J. W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[75] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, 37(1-3):95–138, 1998.

[76] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[77] L. Gaspero, A. Rendl, and T. Urli. Constraint-Based Approaches for Balancing Bike Sharing Systems. In *CP'13, 758–773*. LNCS 8124, Springer, 2013.

[78] Gecode: Generic Constraint Development Environment. Version 3.7.3. `http://www.gecode.org/`.

[79] I. Gent, E. MacIntyre, P. Prosser, B. Smith, and T. Walsh. An Empirical Study of Dynamic Variable Ordering Heuristics for the Constraint Satisfaction Problem. In *CP'96, 179–193*. LNCS 1118. Springer, 1996.

[80] I. P. Gent. Heuristic Solution of Open Bin Packing Problems. *Journal of Heuristics*, 3(4):299–304, 1998.

[81] I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. In *In: Proceedings of ECAI 2006, Riva del Garda*, pages 98–102. IOS Press, 2006.

[82] I. P. Gent and T. Walsh. From Approximate to Optimal Solutions: Constructing Pruning and Propagation Rules. In *International joint conference on Artifical intelligence*, pages 1396–1401, 1997.

[83] M. L. Ginsberg, M. Frank, M. P. Halpin, and M. C. Torrance. Search Lessons Learned from Crossword Puzzles. In *AAAI'90, 210–215*. The MIT Press, 1990.

[84] A. Goldsztejn, L. Granvilliers, and C. Jermann. Constraint Based Computation of Periodic Orbits of Chaotic Dynamical Systems. In *CP'13, 774–789*. LNCS 8124, Springer, 2013.

[85] J. González-Moreno, M. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40:47–87, 1999.

[86] S. Grandcolas and C. Pinto. A SAT Encoding for Multi-dimensional Packing Problems. In *CPAIOR'10, 141–146*. LNCS 6140. Springer, 2010.

[87] S. Gualandi and M. Lombardi. A Simple and Effective Decomposition for the Multidimensional Binpacking Constraint. In *CP'13, 356–364*. LNCS 8124, Springer, 2013.

[88] T. Guns, A. Dries, G. Tack, S. Nijssen, and L. De Raedt. MiningZinc: A Modeling Language for Constraint-based Mining. In *IJCAI'13, 1365–1372*. AAAI Press, 2013.

[89] P. Hansen and N. Mladenovic. Variable Neighborhood Search, 1997.

[90] M. Hanus. A Unified Computation Model for Functional and Logic Programming. In *POPL'97, 80–93*. ACM, 1997.

[91] M. Hanus. Multi-Paradigm Declarative Languages. In *ICLP'07, 45–75*. LNCS 4670. Springer, 2007.

[92] M. Hanus. Curry: An Integrated Functional Logic Language, 2012. `http://www-ps.informatik.uni-kiel.de/currywiki/documentation/report`.

[93] M. Hanus. PAKCS User Manual, 2013. `www.informatik.uni-kiel.de/pakcs`.

[94] R. Haralick and G. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence*, 14:263–313, 1980.

[95] W. D. Harvey and M. L. Ginsberg. Limited Discrepancy Search. In *IJCAI'95, 607–613*. Morgan Kaufmann.

[96] Haskell-Wiki. The Haskell Programming Language, 2014. `http://www.haskell.org/haskellwiki/Haskell`.

[97] J. He, P. Flener, J. Pearson, and W. Zhang. Solving String Constraints: The Case for Constraint Programming. In *CP'13, 381–397*. LNCS 8124, Springer, 2013.

[98] E. Hebrard, E. O'Mahony, and B. O'Sullivan. Constraint programming and combinatorial optimisation in numberjack. In *CPAIOR*, pages 181–185, 2010.

[99] S. M. Heist. *A Comparison of Constraint Programming and Integer Programming for an Industrial Planning Problem*. PhD thesis, Lehigh University, 2003.

[100] P. V. Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[101] R. Herken, editor. *A Half-century Survey on The Universal Turing Machine*. Oxford University Press, Inc., 1988.

[102] M. Heydar, M. E. Petering, and D. R. Bergmann. Mixed Integer Programming for Minimizing the Period of a Cyclic Railway Timetable for a Single Track with Two Train Types. *Computers & Industrial Engineering*, 66(1):171 – 185, 2013.

[103] B. Hnich, I. Miguel, I. P. Gent, and T. Walsh. CSPLib: A Problem Library for Constraints. `http://www.csplib.org/`.

[104] P. Hofstedt and P. Pepper. Integration of Declarative and Constraint Programming. *TPLP*, 7(1-2): pages 93–121, 2007.

[105] C. Holzbaur. *Specification of Constraint Based Inference Mechanism through Extended Unification*. PhD thesis, University of Vienna, 1990.

[106] J. Hwang and D. Mitchell. 2-way vs. d-way Branching for CSP. In *CP'05, 343–357*. LNCS 3709. Springer, 2005.

[107] IBM Academic Initiative. `www.ibm.com/developerworks/university/academicinitiative/`.

[108] ILOG. ILOG OPL Studio 3.7, Reference Manual, 2009.

[109] I. ILOG. Ibm ilog solver user's manual, 2013.

[110] JaCoP. http://www.jacop.eu/.

[111] J. Jaffar and M. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19-20:503–581, 1994.

[112] JSR 331: Constraint Programming API. http://jcp.org/en/jsr/detail?id=331/.

[113] J. K. Karlof. *Integer Programming : Theory and Practice*. Taylor & Francis, 2006.

[114] H. Kjellerstrand. Comparison of CP Systems, 2012. http://web.it.kth.se/~cschulte/events/SweConsNet-2012/hakan.pdf.

[115] R. E. Korf. An Improved Algorithm for Optimal Bin Packing. In *IJCAI'03, 1252–1258*, 2003.

[116] M. Labbé, G. Laporte, and S. Martello. Upper Bounds and Algorithms for the Maximum Cardinality Bin Packing Problem. *European Journal of Operational Research*, 149(3):490 – 498, 2003.

[117] M. Lagerkvist, M. Nordkvist, and M. Rattfeldt. Laser Cutting Path Planning Using CP. In *CP'13, 790–804*. LNCS 8124, Springer, 2013.

[118] K. Leo, C. Mears, G. Tack, and M. Garcia de la Banda. Globalizing Constraint Models. In *CP'13, 432–447*. LNCS 8124, Springer, 2013.

[119] A. Letort, N. Beldiceanu, and M. Carlsson. A Scalable Sweep Algorithm for the cumulative Constraint. In *CP'12, 439–454*. LNCS 7514. Springer, 2012.

[120] J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[121] J. W. Lloyd. Practical Advtanages of Declarative Programming. In *GULP-PRODE (1)*, pages 18–30, 1994.

[122] R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A Demand Driven Computation Strategy for Lazy Narrowing. In *PLILP'93, 184–200*. LNCS 714, 1993.

[123] F. López-Fraguas, M. Rodríguez-Artalejo, and R. Vado-Vírseda. A Lazy Narrowing Calculus for Declarative Constraint Programming. In *PPDP'04, 43–54*. ACM, 2004.

[124] F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A Multiparadigm Declarative System. In *RTA'99, 244–247*. LNCS 1631. Springer, 1999.

[125] M. Loth, M. Sebag, Y. Hamadi, and M. Schoenauer. Bandit-Based Search for Constraint Programming. In *CP'13, 464–480*. LNCS 8124, Springer, 2013.

[126] M. Carlsson et. al. SICStus Prolog User's Manual, 2013. http://sicstus.sics.se/sicstus/docs/latest4/pdf/sicstus.pdf.

[127] A. K. Mackworth. On Reading Sketch Maps. In *IJCAI'77, 598–606*. William Kaufmann, 1977.

[128] B. J. MacLennan. *Functional Programming: Practice and Theory*. Addison-Wesley, 1990.

[129] M. Mann, F. Nahar, H. Ekker, R. Backofen, P. Stadler, and C. Flamm. Atom Mapping with Constraint Programming. In *CP'13, 805–822*. LNCS 8124, Springer, 2013.

[130] M. Marin. *Functional Logic Programming with Distributed Constraint Solving*. PhD thesis, 2000.

[131] M. Marin, T. Ida, and W. Schreiner. CFLP: A Mathematica Implementation of a Distributed Constraint Solving System. In *IMS'99, 23–25*. WIT Press, 1999.

[132] K. Marriot and P. J. Stuckey. *Programming with Constraints*. MIT Press, 1998.

[133] K. Marriott, P. Stuckey, L. Koninck, and H. Samulowitz. An Introduction to MiniZinc, 2013. `ww2.cs.mu.oz.au/~pjs/637/resources/mini-zinc-tute.pdf`.

[134] M. Marte. Towards Constraint-Based School Timetabling. *Annals OR*, 155(1):207–225, 2007.

[135] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

[136] S. Martello and P. Toth. Lower Bounds and Reduction Procedures for the Bin Packing Problem. *Discrete Applied Mathematics*, 28:59–70, 1990.

[137] J.-P. Métivier, P. Boizumault, and S. Loudni. Solving Nurse Rostering Problems Using Soft Global Constraints. In *CP'09, 73–87*. LNCS 5732. Springer, 2009.

[138] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *DAC'01, 530–535*. ACM, 2001.

[139] Mozart Programming System. `http://mozart.github.io/`.

[140] R. Naveh and A. Metodi. Beyond Feasibility: CP Usage in Constrained-Random Functional Hardware Verification. In *CP'13, 823–831*. LNCS 8124, Springer, 2013.

[141] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *Computer Journal*, 7:308–313, 1965.

[142] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. MiniZinc: Towards a Standard CP Modelling Language. In *CP'13, 529–543*. LNCS 4741. Springer, 2007.

[143] M. Newton, D. Pham, W. Tan, M. Portmann, and A. Sattar. Stochastic Local Search Based Channel Assignment in Wireless Mesh Networks. In *CP'13, 832–847*. LNCS 8124, Springer, 2013.

[144] Nurse Rostering Competition. www.kuleuven-kulak.be/nrpcompetition.

[145] R. M. P. Galinier, B. Jaumard and G. Pesant. A Constraint-Based Approach to the Golomb Ruler Problem, 2007. http://www.crt.umontreal.ca/~quosseca/pdf/41-golomb.pdf.

[146] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., 1984.

[147] S. Peyton-Jones. *The Implementation of Functional Programming Languages*. Prentice Hall, Englewood Cliffs, N.J., 1987.

[148] S. Peyton-Jones. Haskell 98 Language and Libraries: the Revised Report. Technical report, 2002. http://www.haskell.org/onlinereport/.

[149] M. Pinedo. *Planning and Scheduling in Manufacturing and Services*. Springer Series in Operations Research, 2004.

[150] S. Prolog. SWI Prolog Reference Manual, 2013. http://www.swi-prolog.org/pldoc/refman/.

[151] R. Qu, E. Burke, B. McCollum, L. T. Merlot, and S. Y. Lee. A survey of Search Methodologies and Automated Approaches for Examination Timetabling. 2006.

[152] R. Qu and F. He. A Hybrid Constraint Programming Approach for Nurse Rostering Problems. In T. Allen, R. Ellis, and M. Petridis, editors, *Applications and Innovations in Intelligent Systems XVI*, pages 211–224. Springer, 2009.

[153] R. González-del-Campo and F. Sáenz-Pérez. Programmed Search in a Timetabling Problem over Finite Domains. In *ENTCS, 177, 253–267*. Elsevier, 2007.

[154] J. C. Régin. Global Constraints and Filtering Algorithms. In M. Milano, editor, *Constraints and Integer Programming Combined*.

[155] J.-C. Régin and M. Rezgui. Discussion About Constraint Programming Bin Packing Models. In *Proceedings of the AIDC Workshop*, 2011.

[156] J.-C. Régin, M. Rezgui, and A. Malapert. Embarrassingly Parallel Search. In *CP'13, 596–610*. LNCS 8124, Springer, 2013.

[157] R. M. Reischuk, C. Schulte, P. J. Stuckey, and G. Tack. Maintaining State in Propagation Solvers. In *CP'09, 692–706*. LNCS 5732. Springer, 2009.

[158] V. Rici. *Fitting distribution with R*, 2005. http://cran.r-project.org/doc/contrib/Ricci-distributions-en.pdf.

[159] B. Rieck. Bin packing heuristics. http://bastian.rieck.ru/uni/bin_packing/bin_packing.pdf.

[160] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *J. ACM*, 12(1):23–41, 1965.

[161] M. Rodríguez-Artalejo. Functional and Constraint Logic Programming. In *CCL'99, 202–270*. LNCS 2002. Springer, 2001.

[162] E. Rönnberg and T. Larsson. Automating the Self-Scheduling Process of Nurses in Swedish Healthcare: A Pilot Study. *Health Care Management Science*, 13(1):35–53, 2010.

[163] F. Rossi, P. v. Beek, and T. Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., 2006.

[164] H. G. Santos, T. Toffolo, S. Ribas, and R. Gomes. Integer Programming Techniques for the Nurse Rostering Problem. In *PATAT'12*. Under consideration for publication in Special Issue of the Annals of Operations Research, 2012.

[165] S. Sapkal and D. Laha. A Heuristic for No-wait Flow Shop Scheduling. *The International Journal of Advanced Manufacturing Technology*, 68(5-8), 2013.

[166] P. Schaus. *Solving Balancing and Bin-Packing Problems with Constraint Programming*. PhD thesis, Université Catholique de Louvain-la-Neuve, 2009.

[167] A. Scholl and R. Klein. Bin packing benchmark. http://www.wiwi.uni-jena.de/Entscheidung/binpp/.

[168] A. Scholl, R. Klein, and C. Jürgens. BISON: A Fast Hybrid Procedure for Exactly Solving the One-Dimensional Bin Packing Problem. *Computers & Operations Research*, 24(7):627–645, 1997.

[169] T. Schrijvers, P. Stuckey, and P. Wadler. Monadic Constraint Programming. *Journal of Functional Programming*, 19(6):663–697, 2009.

[170] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, and P. Stuckey. Search combinators. *Constraints*, 18(2):269–305, 2013.

[171] T. Schrijvers, M. Triska, and B. Demoen. Tor: Extensible Search with Hookable Disjunction. In *PPDP'12, 103–114*. ACM, 2012.

[172] C. Schulte, G. Tack, and M. Z. Lagerkvist. Gecode FlatZinc interpreter, 2013. http://www.gecode.org/flatzinc.html.

[173] C. Schulte, G. Tack, and M. Z. Lagerkvist. Modeling and Programming with Gecode, 2014. http://www.gecode.org/doc-latest/MPG.pdf.

[174] A. Schutt, T. Feydy, P. Stuckey, and M. Wallace. Solving RCPSP/max by Lazy Clause Generation. *Journal of Scheduling*, 16(3):273–289, 2013.

[175] P. Schwerin and G. Wäscher. The Bin-Packing Problem: A Problem Generator and Some Numerical Experiments with FFD Packing and MTP. *International Transactions in Operational Research*, 4(5-6):377–389, 1997.

[176] P. Schwerin and G. Wäscher. *A New Lower Bound for the Bin-Packing Problem and its Integration Into MTP*. Martin-Luther-Univ., 1998.

[177] P. Shaw. A Constraint for Bin Packing. In *CP'04, 648–662*. LNCS 3258. Springer, 2004.

[178] SICStus Prolog. `http://www.sics.se/`.

[179] G. Sierksma, P. v. Dam, and G. A. Tijssen. *Linear and Integer Programming : Theory and Practice*. Dekker, 1996.

[180] H. Simonis and B. O'Sullivan. Search Strategies for Rectangle Packing. In *CP'08, 52–66*. LNCS 5202. Springer, 2008.

[181] H. Simonis and B. O'Sullivan. Almost Square Packing. In *CPAIOR'11, 196–209*. LNCS 6697. Springer, 2011.

[182] P. Simons. Smodels, 2008. `http://www.tcs.hut.fi/Software/smodels/`.

[183] B. M. Smith, K. Stergiou, and T. Walsh. Modelling the Golomb Ruler Problem, 1999.

[184] Z. Somogyi, F. J. Henderson, and T. C. Conway. Mercury, an Efficient Purely Declarative Logic Programming Language. In *Australian Computer Science Conference*, pages 499–512, 1995.

[185] K. C. Sou, J. Weimer, H. Sandberg, and K. H. Johansson. Scheduling Smart Home Appliances Using Mixed Integer Linear Programming. In *CDC-ECE'11, 5144–5149*. IEEE, 2011.

[186] M. R. Steenberger. Maximum Likelihood Programming in R. `http://www.unc.edu/~monogan/computing/r/MLE_in_R.pdf`.

[187] P. Stuckey. Lazy Clause Generation: Combining the Power of SAT and CP (and MIP?) Solving. In *CPAIOR'10, 5–9*. LNCS 6140. Springer, 2010.

[188] P. J. Stuckey and H. Kjellerstrand. First International Lightning Model and Solve Competition, 2013. `http://cp2013.a4cp.org/program/competition`.

[189] P. J. Stuckey and G. Tack. Minizinc with Functions. In *CPAIOR'13, 268–283*. LNCS 7874, Springer, 2013.

[190] SWI. `www.swi-prolog.org/`.

[191] A. Tomás and J. Leal. Automatic Generation and Delivery of Multiple-Choice Math Quizzes. In *CP'13, 848–863*. LNCS 8124, Springer, 2013.

[192] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[193] P. Van Hentenryck. *The OPL Optimization Programming Language*. MIT Press, 1999.

[194] P. Van Roy, P. Brand, D. Duchier, S. Haridi, M. Henz, and C. Schulte. Logic Programming in the Context of Multiparadigm Programming: The Oz Experience. *TPLP*, 3(6):717–763, 2003.

[195] A. Vidotto. Online Constraint Solving and Rectangle Packing. In *CP'04, 807*. LNCS 3258. Springer, 2004.

[196] P. Vismara, R. Coletta, and G. Trombettoni. Constrained Wine Blending. In *CP'13, 864–879*. LNCS 8124, Springer, 2013.

[197] M. Wallace, J. Schimpf, K. Shen, and W. Harvey. On Benchmarking Constraint Logic Programming Platforms. *Constraints*, 9(1):5–34, 2004.

[198] T. Walsh. Symmetry Breaking Constraints: Recent Results. In *AAAI*, 2012.

[199] G. Wäscher and T. Gau. Heuristics for the Integer One-Dimensional Cutting Stock Problem: A Computational Study. *OR Spectrum*, 18(3):131–144, 1996.

[200] T. V. Weelden. Nurse Rostering through Linear Programming and Repair Heuristics, 2013. Master's Thesis, University of Utrecht, the Netherlands.

[201] W. Weibull. A Statistical Distribution Function of Wide Applicability. *Journal of Applied Mechanics Transactions*, 18(3):293–297, 1951.

[202] P. Wessa. Free Statistics Software, Office for Research Development and Education. Version 1.1.23-r7. `http://www.wessa.net/`.

[203] J. Wielemaker. 25 Years of SWI-Prolog, 2012. `www.swi-prolog.org/download/publications/ICLP12-SWI25.odp`.

[204] T.-H. Wu, J.-Y. Yeh, and Y.-M. Lee. A Particle Swarm Optimization Approach for Nurse Rostering Problem. In *BAI'13*, 2013.

[205] P. Wuille and T. Schrijvers. Parameterized Models for On-Line and Off-Line Use. In *WFLP'10, 171–185*. LNCS 6559. Springer, 2010.

[206] YACS. `http://constraints.sourceforge.net/`.

[207] M. Yang, L. Cai, and G. Song. Constraint Satisfaction Timetabling Research Based on Course-Period-Template Selection and Conflict-Vector Detection. In *FCST'10, 582–588*, 2010.

[208] S. Zampelli, Y. Vergados, R. Schaeren, W. Dullaert, and B. Raa. The Berth Allocation and Quay Crane Assignment Problem Using a CP Approach. In *CP'13, 880–896*. LNCS 8124, Springer, 2013.

Resumen en español

(Summary in Spanish)

La presente tesis doctoral ha sido redactada completamente en inglés. Por ello, y siguiendo la normativa de la Universidad Complutense de Madrid, se incluye una versión en castellano del *Abstract* ("Compendio") y las *Keywords* ("Palabras clave"), así como un amplio resumen de la tesis, con una sección dedicada a cada una de las cinco partes en que se esta se divide: *Introduction and Preliminaries* ("Introducción y preliminares"), *Improving the Performance of* $\mathcal{TOY}(\mathcal{FD})$ ("Mejora del rendimiento de $\mathcal{TOY}(\mathcal{FD})$"), *Real-life Applications of* $\mathcal{TOY}(\mathcal{FD})$ ("Aplicaciones reales de $\mathcal{TOY}(\mathcal{FD})$"), *Positioning* $\mathcal{TOY}(\mathcal{FD})$ *w.r.t. Other CP(*$\mathcal{FD}$*) Systems* ("$\mathcal{TOY}(\mathcal{FD})$ comparado con otros sistemas CP($\mathcal{FD}$)") y *Conclusions and Perspectives* ("Conclusiones y trabajo futuro").

Compendio: Debido a su importancia en la industria, los problemas de satisfacción y optimización de restriccciones (CSP y COP, respectivamente) han sido ampliamente estudiados en las últimas décadas. Estos implican una compleja formulación y un gran esfuerzo computacional para su resolución. El área de conocimiento de la programación con restricciones sobre dominios finitos (CP($\mathcal{FD}$)) ha sido identificada como especialmente adecuada para el modelado y resolución de un CSP o COP, ya que captura la naturaleza orientada a restricciones de estos problemas de una manera concisa. Dentro de CP($\mathcal{FD}$), los cuatro paradigmas CP($\mathcal{FD}$) algebraico, C++ CP($\mathcal{FD}$), programación lógica con restricciones (CLP($\mathcal{FD}$)) y programación lógico funcional con restriccciones (CFLP($\mathcal{FD}$)) se basan en un resolutor de restricciones, pero difieren en el lenguaje de modelado utilizado. En particular, CFLP($\mathcal{FD}$) ofrece lenguajes altamente expresivos, soportando e incluso incrementando las características de modelado de los lenguajes lógicos y funcionales. Sin embargo, aunque CFLP($\mathcal{FD}$) resulta ser un paradigma adecuado para hacer frente a los CSP y COP, la literatura carece de tantas aplicaciones como las existentes para CP($\mathcal{FD}$) algebraico, C++ CP($\mathcal{FD}$) y CLP($\mathcal{FD}$).

El objetivo principal de esta tesis es fomentar el uso de CFLP($\mathcal{FD}$) para hacer frente a CSP y COP de la vida real. Para ello, la investigación se divide en tres partes: una mejora del rendimiento de CFLP($\mathcal{FD}$), una descripción de aplicaciones reales para CFLP($\mathcal{FD}$) y una comparativa en profundidad sobre el modelado y resolución de varios COP utilizando sistemas CP($\mathcal{FD}$) algebraicos, C++ CP($\mathcal{FD}$), CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$). Para ello, se selecciona al sistema CFLP($\mathcal{FD}$) de vanguardia $\mathcal{TOY}(\mathcal{FD})$, implementado en SICStus Prolog, y que soporta la resolución de restricciones $\mathcal{FD}$ mediante su interacción con un resolutor de restricciones.

La primera parte de la investigación mejora el rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD})$. En concreto, se desarrolla un esquema genérico para integrar resolutores C++ CP($\mathcal{FD}$) en $\mathcal{TOY}(\mathcal{FD})$, y se implementan dos nuevas versiones del sistema que resultan de instanciar el esquema con Gecode e ILOG Solver. Asimismo, se aumenta la capacidad expresiva de $\mathcal{TOY}(\mathcal{FD})$ con nuevas primitivas de búsqueda. Estas permiten una mejor especificación de estrategias de búsqueda *ad hoc*, que explotan la estructura del problema a resolver, precisando una menor exploración de búsqueda para computar las soluciones del mismo.

Una vez mejorado el rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD})$, la segunda parte de la investigación presenta dos aplicaciones industriales del sistema. La primera es un problema de asignación de trabajadores a turnos de trabajo (ETP), proveniente de la industria de las comunicaciones, y modelado y resuelto con $\mathcal{TOY}(\mathcal{FD})$. La segunda es un análisis empírico de la complejidad del problema de asignación de elementos unidimensionales a contenedores (BPP). En este análisis se utilizan tanto técnicas heurísticas como CP($\mathcal{FD}$) (esta última incluyendo a $\mathcal{TOY}(\mathcal{FD})$) para resolver un conjunto de casos de prueba, que resulta representativo de instancias generalizadas del problema provenientes de la industria de la optimización en centros de datos.

Por último, la tercera parte de la investigación utiliza a ETP para realizar una compa-

rativa en profundidad del modelado y resolución del problema en diferentes sistemas CP($\mathcal{FD}$) de vanguardia. En concreto, se seleccionan los sistemas CP($\mathcal{FD}$) algebraicos Minizinc e ILOG OPL, los sistemas C++ CP($\mathcal{FD}$) Gecode e ILOG Solver, los sistemas CLP($\mathcal{FD}$) SICStus Prolog y SWI-Prolog, y los sistemas CFLP($\mathcal{FD}$) PAKCS y $\mathcal{TOY}(\mathcal{FD})$. La comparativa muestra que $\mathcal{TOY}(\mathcal{FD})$ es competitivo con respecto a cualquiera de los otros sistemas.

Palabras clave: Programación con restricciones sobre dominios finitos, programación lógico funcional con restricciones, integración de resolutores de restricciones, estrategias de búsqueda, problema de asignación de trabajadores a turnos de trabajo, problema de asignación de elementos unidimensionales a contenedores, generación paramétrica de casos de prueba, programación con restricciones algebraica, programación con restricciones orientada a objetos, programación lógica con restricciones.

1 Introducción y preliminares

La evolución tecnológica, influenciada en parte por las necesidades económicas, ha convertido a la logística en un factor fundamental para el éxito de cualquier empresa u organización. Detrás de esta idea abstracta se encuentran problemas concretos cuya naturaleza combinatoria a menudo los hace NP-completos (es decir, problemas sobre los que no se pueden aplicar métodos generales de coste polinómico) [76]. Por lo tanto, se requiere tiempo y experiencia, tanto para la especificación del problema como el diseño del algoritmo que lo resuelva.

1.1 CSP y COP

Los *problemas de satisfacción y optimización de restriccciones* (del inglés *Constraint Satisfaction and Optimization Problem: CSP* y *COP*, respectivamente) [192] proporcionan una formalización abstracta para los problemas relacionados con la asignación de recursos. Están presentes en las industrias manufactureras y de servicios, tales como la producción, el transporte, la distribución, el procesamiento de la información y las comunicaciones [149]. Un CSP se define mediante la tupla (V, D, C), donde V es el conjunto de variables $\{v_1, \ldots, v_n\}$, D es el conjunto de n dominios $\{d_1, \ldots, d_n\}$ (donde cada d_i representa los posibles valores que v_i puede tomar) y C es el conjunto de restricciones (cada una de ellas involucrando a un subconjunto de variables de V, y proporcionando una representación intensional de las combinaciones de valores satisfactibles que estas variables pueden tomar). Una solución del problema es una asignación de variables V a valores de D, de modo que las restricciones de C se satisfagan. Un ejemplo paradigmático de un CSP es el problema de las N *reinas*: un puzzle cuyo desafío consiste en colocar N reinas de ajedrez en un tablero de $N \times N$ casillas, de modo que no haya dos reinas que se ataquen entre sí (es decir, que no haya dos reinas colocadas en la misma fila, columna o diagonal).

Por otro lado, un COP se define mediante la tupla (V, D, C, F), donde el parámetro adicional F representa una función de coste (es decir, una expresión a minimizar o maximizar). Una solución del problema es una asignación de variables V a valores de D, de modo que las restricciones de C se satisfagan y la función de coste F se minimice o maximice. Un ejemplo paradigmático de un COP es el problema de las *reglas con* N *marcas de Golomb*: un puzzle cuyo desafío consiste en dibujar N marcas ($0 = m_0 < m_1 < \ldots < m_{N-1}$) en una regla, de modo que las distancias $d_{i,j} = m_j - m_i$ para $0 \leq i < j < N$ sean distintas. Una regla óptima es la de longitud mínima (minimizando así el valor de la última marca m_{N-1}).

A modo de ejemplo, la figura 1 presenta una posible especificación de la instancia G-5 (para el problema de Golomb con cinco marcas en la regla), incluyendo el espacio inicial de búsqueda del problema y remarcando en negrita su solución óptima, para la cual también se proporciona una representación gráfica.

$V \equiv [m_0, m_1, m_2, m_3, m_4,$
$d_{1,0}, d_{2,0}, d_{3,0}, d_{4,0}, d_{2,1}, d_{3,1}, d_{4,1}, d_{3,2}, d_{4,2}, d_{4,3}]$;
$D \equiv [m_0$ in $0..15, m_1$ in $0..15, m_2$ in $0..15, m_3$ in $0..15, m_4$ in $0..15,$
$d_{1,0}$ in $0..15, d_{2,0}$ in $0..15, d_{3,0}$ in $0..15, d_{4,0}$ in $0..15, d_{2,1}$ in $0..15,$
$d_{3,1}$ in $0..15, d_{4,1}$ in $0..15, d_{3,2}$ in $0..15, d_{4,2}$ in $0..15, d_{4,3}$ in $0..15]$;
$C \equiv [m_0 = 0,$
$m_0 < m_1, m_1 < m_2, m_2 < m_3, m_3 < m_4,$
$d_{1,0} = m_1 - m_0, d_{2,0} = m_2 - m_0, d_{3,0} = m_3 - m_0, d_{4,0} = m_4 - m_0,$
$d_{2,1} = m_2 - m_1, d_{3,1} = m_3 - m_1, d_{4,1} = m_4 - m_1,$
$d_{3,2} = m_3 - m_2, d_{4,2} = m_4 - m_2, d_{4,3} = m_4 - m_3$
$d_{1,0} != d_{2,0}, d_{1,0} != d_{3,0}, d_{1,0} != d_{4,0}, d_{1,0} != d_{2,1}, d_{1,0} != d_{3,1}, d_{1,0} != d_{4,1},$
$d_{1,0} != d_{3,2}, d_{1,0} != d_{4,2},$
$d_{1,0} != d_{4,3}, d_{2,0} != d_{3,0}, d_{2,0} != d_{4,0}, d_{2,0} != d_{2,1}, d_{2,0} != d_{3,1}, d_{2,0} != d_{4,1},$
$d_{2,0} != d_{3,2}, d_{2,0} != d_{4,2},$
$d_{2,0} != d_{4,3}, d_{3,0} != d_{4,0}, d_{3,0} != d_{2,1}, d_{3,0} != d_{3,1}, d_{3,0} != d_{4,1}, d_{3,0} != d_{3,2},$
$d_{3,0} != d_{4,2}, d_{3,0} != d_{4,3},$
$d_{4,0} != d_{2,1}, d_{4,0} != d_{3,1}, d_{4,0} != d_{4,1}, d_{4,0} != d_{3,2}, d_{4,0} != d_{4,2}, d_{4,0} != d_{4,3},$
$d_{2,1} != d_{3,1}, d_{2,1} != d_{4,1},$
$d_{2,1} != d_{3,2}, d_{2,1} != d_{4,2}, d_{2,1} != d_{4,3}, d_{3,1} != d_{4,1}, d_{3,1} != d_{3,2}, d_{3,1} != d_{4,2},$
$d_{3,1} != d_{4,3}, d_{4,1} != d_{3,2},$
$d_{4,1} != d_{4,2}, d_{4,1} != d_{4,3}, d_{3,2} != d_{4,2}, d_{3,2} != d_{4,3}, d_{4,2} != d_{4,3},$
$d_{1,0} >= 1, d_{2,0} >= 3, d_{3,0} >= 6, d_{4,0} >= 10, d_{2,1} >= 1, d_{3,1} >= 3,$
$d_{4,1} >= 6, d_{3,2} >= 1, d_{4,2} >= 3, d_{4,3} >= 1$
$d_{1,0} < d_{4,3}]$;
$F \equiv$ minimize m_4;

$S \equiv [(m_0 = 0, m_1 = 0, m_2 = 0, m_3 = 0, m_4 = 0, d_{1,0} = 0, d_{2,0} = 0, d_{3,0} = 0, \ldots d_{4,3} = 0),$
$\ldots, (m_0 = \mathbf{0}, m_1 = \mathbf{1}, m_2 = \mathbf{4}, m_3 = \mathbf{9}, m_4 = \mathbf{11}, d_{1,0} = \mathbf{1}, d_{2,0} = \mathbf{4},$
$d_{3,0} = \mathbf{9}, d_{4,0} = \mathbf{11}, d_{2,1} = \mathbf{3}, d_{3,1} = \mathbf{8}, d_{4,1} = \mathbf{10}, d_{3,2} = \mathbf{5}, d_{4,2} = \mathbf{7}, d_{4,3} = \mathbf{2}), \ldots$
$(m_0 = 15, m_1 = 15, m_2 = 15, m_3 = 15, m_4 = 15, d_{1,0} = 15, \ldots d_{4,3} = 15)]$

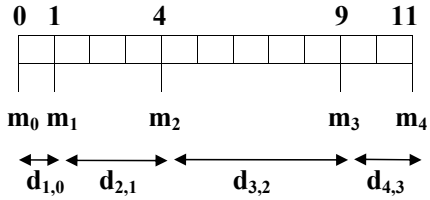

Figura 1. Especificación y solución óptima para G-5

Además de utilizar N variables para representar las marcas de la regla $[m_0, \ldots m_{N-1}]$, se usan $N*(N-1)/2$ variables adicionales para representar las distancias entre cada par de marcas $[d_{1,0}, \ldots, d_{(N-1),0}, d_{2,1}, \ldots, d_{(N-1),(N-2)}]$ (dando lugar a un total de $(N^2+N)/2$ variables). Es sabido que se puede aplicar la cota superior $2^{N-1}-1$ sobre el valor que pueden tomar las variables m [145]. Esto se basa en que, si la distancia entre m_j y m_{j+1} es siempre 2^{j+1}, entonces cada m_i se puede asignar a 2^i-1. Por lo tanto, la representación en formato de bits de cualquier $d_{j,i}$ = m_j - m_i toma el valor cero en sus i bits más pequeños, seguidos de $j-i$ unos. Como se puede ver, esta representación en formato de bits es diferente para cada una de las variables de d, por lo que sus valores son diferentes también [173]. Estableciendo que las variables m tomen un dominio inicial de $0..(2^{N-1}-1)$, entonces la distancia entre cada par de marcas debe estar también en este dominio, por lo que las variables d se acotan igualmente a $0..(2^{N-1}-1)$. El resto de requisitos del problema se especifican mediante las siguientes N^2 restricciones: se usa una restricción para asignar m_0 a 0, y $N-1$ restricciones especifican que cada m_i sea menor que m_{i+1}. Otras $N*(N-1)/2$ restricciones asignan cada $d_{i,j}$ a la resta de sus dos marcas asociadas. Por último, otras tantas restricciones especifican que todas las variables d sean diferentes.

Además de estas N^2 restricciones, la especificación de Golomb-5 contiene ciertas restricciones redundantes que mejoran el rendimiento a la hora de encontrar las soluciones del problema. Por ejemplo, se utilizan $N*(N-1)/2$ restricciones redundantes para aplicar una cota inferior a las variables d [183]. Esto se basa en que la distancia $d_{j,i}$ = m_j - m_i satisface la propiedad de ser igual a la suma de todas las distancias entre las marcas m_i y m_j (es decir, $d_{i,j}$ = (m_j - m_{j-1}) + (m_{j-1} - m_{j-2}) + ... + (m_{i+1} - m_i)). Como todas estas distancias son distintas, su suma debe ser, al menos, la suma de los primeros $j-i$ números naturales. Del mismo modo, se añade una última restricción redundante $d_{1,0} < d_{(N-2),(N-1)}$ para romper algunas simetrías entre las soluciones encontradas.

Finalmente, la función de coste especifica que, entre todas las soluciones satisfactibles del problema, solo se debe tener en cuenta aquella (o aquellas) con un valor mínimo para m_4. Es decir, la naturaleza combinatoria del problema da lugar a un espacio de búsqueda de $(2^{N-1})^{(N^2+N)/2}$ candidatos a solución (ya que el problema contiene $(N^2+N)/2$ variables, cada una de ellas con un dominio inicial de 2^{N-1} valores). Incluso en una instancia del problema tan pequeña como es Golomb-5, esta combinatoria conduce a una cantidad de $16^{15} \equiv 1.152.921.504.606.846.976$ candidatos, de los que solo 81 son soluciones satisfactibles. Analizándolas con respecto al valor que toman para m_4: 38 soluciones asignan m_4 al valor 15, 21 soluciones a 14, 14 soluciones a 13, 7 soluciones a 12, y, por último, solo 1 solución a 11 (siendo esta, por tanto, la solución óptima).

1.2 CP($\mathcal{FD}$) para abordar un CSP o COP

Los CSP y COP han sido ampliamente estudiados en las últimas décadas. Para su tratamiento se han aplicado diferentes aproximaciones: el área de conocimiento de la *programación matemática* (del inglés *Mathematical Programming: MP*) ha aplicado tanto técnicas de *programación lineal* (del inglés *Linear Programming: LP*) [179] como técnicas de *programación entera (mixta)* (del inglés *(Mixed) Integer Programming: MIP*) [113]. Cuando la aplicación de una búsqueda exhaustiva resultaba impracticable (debido al enorme espacio de búsqueda del problema), el área de conocimiento de las *heurísticas* ha aplicado aproximaciones basadas en una búsqueda incompleta, como *estrategias ad hoc* [146] o incluso *meta heurísticas* [39] (proporcionando heurísticas para determinar la heurística finalmente aplicada para resolver el problema).

Además de las mencionadas áreas MP y heurísticas, el área de conocimiento de *programación con restricciones sobre dominios finitos* (del inglés *Constraint Programming over Finite Domains: CP($\mathcal{FD}$)*) [63], [163] ha sido identificada como especialmente adecuada para el modelado y resolución de un CSP o COP, ya que captura la naturaleza orientada a restricciones de estos problemas de una manera concisa. Para ello, CP($\mathcal{FD}$) distingue entre el lenguaje de modelado utilizado para especificar el problema y las técnicas aplicadas para resolverlo.

Cualquier sistema CP($\mathcal{FD}$) se basa en la noción de un *resolutor de restricciones* $\mathcal{FD}$. Este resulta de la combinación de un *almacén de restricciones* y un *motor de restricciones*. El almacén contiene las variables, dominios, restricciones y posible función de coste ((V, D, C) y (V, D, C, F)) del CSP o COP especificado. El motor aplica técnicas tanto de *propagación de restricciones* como de *búsqueda* para encontrar soluciones satisfactibles y óptimas para el problema. Brevemente, la propagación de una restricción c, involucrando a las variables $[v_{i1}, \ldots, v_{ik}]$, es un proceso de inferencia que permite eliminar valores (o combinaciones de valores) del dominio de las variables que no satisfacen la restricción. Como este mecanismo considera cada restricción de la red C por separado, no todos los valores que permanecen en los dominios son necesariamente parte de alguna solución. Así, el proceso de resolución se debe completar con una búsqueda que modifica el CSP o COP inicial mediante la adición de nuevas restricciones para razonar nuevamente sobre él. Las exploraciones de búsqueda sistemáticas también se llaman *algoritmos de búsqueda con vuelta atrás*, y se puede ver como la exploración de un árbol mediante un recorrido de *primero en profundidad* (del inglés *Depth First Search: DFS*), donde cada nodo contiene una versión modificada del CSP o COP original. Un nodo del árbol puede representar: un CSP resuelto (en cuyo caso, la búsqueda se detiene, devolviendo la solución), un CSP insatisfactible (en cuyo caso, el proceso de búsqueda retrocede en el árbol), o un CSP estable, pero con ciertas variables aún sin asignar a valores concretos (en cuyo caso, se selecciona una variable no asignada, y los hijos del nodo se generan mediante la asignación exclusiva y exhaustiva de la variable a los distintos valores de su dominio).

El modo en que se especifican las variables, dominios, restricciones y función de coste depende del lenguaje de modelado utilizado. Dependiendo del contexto en el que se modela el problema, la importancia de diferentes factores (como la expresividad, la aumento de la complejidad del problema, el mantenimiento, la integración en aplicaciones de mayor tamaño, etc.) varía. La mayoría de estas tareas son mutuamente excluyentes y, por lo tanto, cualquier lenguaje de modelado proporciona un compromiso entre ellos. Para presentar al conjunto de paradigmas CP($\mathcal{FD}$) identificados como adecuados para abordar a un CSP o COP, una primera clasificación permite distinguir a los *lenguajes declarativos* de los *lenguajes imperativos*. Un programa declarativo describe las propiedades que una solución del problema debe cumplir, mientras que un programa imperativo describe una secuencia de pasos que se deben realizar con el fin de construir la solución del problema.

La integración de CP($\mathcal{FD}$) en el paradigma imperativo de la *programación orientada a objetos* [33], más específicamente en el lenguaje C++, ha dado lugar al paradigma *C++ CP($\mathcal{FD}$)*. En él, la especificación del problema se beneficia de características de modelado tales como la abstracción, la encapsulación, la herencia y el polimorfismo. Además, la alta eficiencia de C++ [36] permite a las bibliotecas CP($\mathcal{FD}$) implementadas en dicho lenguaje obtener un mayor rendimiento de resolución. Dos sistemas C++ CP($\mathcal{FD}$) de vanguardia son Gecode [78] e IBM ILOG Solver [12].

Dentro de la programación declarativa, una segunda clasificación permite distinguir a los lenguajes de propósito general (también llamados Turing completos [101]) de los lenguajes de propósito específico (o Turing incompletos). La integración de CP($\mathcal{FD}$) en lenguajes de propósito específico basados en formulaciones algebraicas ha dado lugar al paradigma *CP($\mathcal{FD}$) algebraico*. En él, la especificación del problema se beneficia de características de modelado tales como la combinación de restricciones básicas para construir nuevas restricciones complejas, la definición de nuevas restricciones mediante predicados, el uso de tipos enumerados, el uso de estructuras de datos basados en arrays o conjuntos, y en un aislamiento entre el modelo generico y los datos de entrada de la instancia concreta a ejecutar. Además, se abstrae la especificación del CSP o COP del resolutor de restricciones concreto que se utilizará para su resolución, permitiendo así probar un mismo modelo sobre distintos resolutores sin ningún desarrollo adicional. Dos sistemas CP($\mathcal{FD}$) algebraicos de vanguardia son MiniZinc [142] e IBM ILOG OPL [193].

Dentro de los lenguajes declarativos de propósito general, la integración de CP($\mathcal{FD}$) en los lenguajes de *programación lógica* LP [120] ha dado lugar al paradigma de la *programación lógica con restricciones: CLP($\mathcal{FD}$)* [111]. En él, la especificación del problema se beneficia de características de modelado tales como la alta expresividad, incluyendo características lógicas como la notación relacional, el indeterminismo, la vuelta atrás (del inglés *backtracking*), las variables lógicas y de dominio, así como la capacidad para efectuar un *razonamiento por modelos* (permitiendo añadir y eliminar dinámicamente restricciones del almacén). Dos sistemas CLP($\mathcal{FD}$) de vanguardia son

SICStus Prolog [178] y SWI-Prolog [190], cada uno de ellos integrando CP($\mathcal{FD}$) a través de la biblioteca subyacente `clpfd`.

1.3 CFLP($\mathcal{FD}$) para abordar a un CSP o COP

La integración de CP($\mathcal{FD}$) en lenguages declarativos de naturaleza multi-paradigma, como son los lenguajes de *programación lógico funcionales* (del inglés *Functional Logic Programming: FLP*) [161], [91], [19] (que resultan de la integración de LP y la *programación funcional* (del inglés *Functional Programming: FP*) [128]), ha dado lugar al paradigma de la *programación lógico funcional con restricciones* (del inglés *Constraint Functional Logic Programming: CFLP($\mathcal{FD}$)*). En términos de modelado, el lenguaje proporcionado por CFLP($\mathcal{FD}$) representa probablemente el enfoque más completo dentro de los sistemas CP($\mathcal{FD}$). En primer lugar, su naturaleza declarativa abstrae la especificación del problema, lo que representa una ventaja con respecto a los sistemas C++ CP($\mathcal{FD}$) imperativos. En segundo lugar, su naturaleza de propósito general permite la integración del modelo en grandes aplicaciones, en contraste con los sistemas CP($\mathcal{FD}$) algebraicos. Finalmente, su alta expresividad aumenta incluso la de los sistemas CLP($\mathcal{FD}$), incluyendo características propias de FP, tales como la notación funcional, las expresiones *currificadas*, las funciones de orden superior, los patrones, las aplicaciones parciales, la evaluación perezosa, los tipos, el polimorfismo y la composición de funciones. Dos sistemas CFLP($\mathcal{FD}$) de vanguardia son PAKCS [93] (una de las implementaciones del lenguaje Curry [92]) y $\mathcal{TOY}(\mathcal{FD})$ [72], [124], [40].

A modo de ejemplo, la figura 2 presenta el modelo $\mathcal{TOY}(\mathcal{FD})$ para especificar el problema de Golomb de la sección 1.1, evidenciando la alta expresividad de CFLP($\mathcal{FD}$). Basándose en dicho modelo la siguiente sesión $\mathcal{TOY}(\mathcal{FD})$ computa la solución óptima para la instancia G-5:

```
TOY(FD)> golomb 5 == L
      { L -> [ 0, 1, 4, 9, 11 ] }
sol.1, more solutions (y/n/d/a) [y]?
      no
```

El modelo incluye los archivos `cflpfd.toy` y `misc.toy` (líneas 1 y 2, respectivamente). El primero permite el uso de las restricciones $\mathcal{FD}$. El segundo contiene un preludio que incluye una versión $\mathcal{TOY}(\mathcal{FD})$ de diferentes funciones primitivas FP. Se utilizan las funciones `map`, `foldl`, `zipWith`, `scanl`, `iterate`, `head`, `last` y `take`, así como los operadores `(++)` y `/\`. Todos ellos tienen la misma semántica que su versión estándar en FP.

La función principal `golomb` (líneas 3-12) modela el COP. Su única regla recibe el parámetro de entrada `N` que representa la cantidad de marcas a colocar en la regla. La función computa como resultado la lista `M -> ` $[M_0, \ldots, M_{N-1}]$, donde M_i representa la *i*-ésima marca de la regla. Para calcular la solución óptima, la regla se convierte

```
%---------------------------------
% ARCHIVOS INCLUIDOS
%---------------------------------
(01) include "cflpfd.toy"
(02) include "misc.toy"
%---------------------------------
% FUNCION PRINCIPAL
%---------------------------------
(03) golomb :: int -> [int]
(04) golomb N = M <==
(05)    M == take N [0 | gen_v_list],
(06)    order (M ++ [trunc(2^(N-1))]) == true,
(07)    gen_difs M == Ds,
(08)    foldl (++) [] Ds == D,
(09)    all_different D,
(10)    lbound Ds sums_nats == true,
(11)    (head D) #< (last D),
(12)    labeling [toMinimize (last M)] M

%---------------------------------
% FUNCIONES EXTRA
%---------------------------------
(13) order:: [int]  -> bool
(14) order [X] = true
(15) order [X,Y|Xs] = (X #< Y) /\ order [Y|Xs]
%
(16) gen_difs:: [int] -> [[int]]
(17) gen_difs [] = []
(18) gen_difs [X|Xs] = [map (#- X) Xs|gen_difs Xs]
%
(19) lbound:: [[int]] -> [int] -> bool
(20) lbound Xss Is = foldl (/\) true
         (foldl (++) [] (map (zipWith (#<=) Is) Xss))
%
(21) sums_nats:: [int]
(22) sums_nats = scanl (+) 1 (iterate (+1) 2)
%
(23) gen_v_list:: [A]
(24) gen_v_list = [X | gen_v_list]
```

Figura 2. Modelo $\mathcal{TOY}(\mathcal{FD})$ para Golomb

en condicional, con ocho condiciones (líneas 5-12) que se deben satisfacer:

- La línea 5 genera la lista `M`, mediante el uso de la función extra `gen_v_list` (líneas 23-24). Esta lista está compuesta por un 0 (garantizando el requisito del problema de que M_0 = 0) y `N-1` nuevas variables lógicas.
- La línea 6 utiliza la función extra `order` (líneas 13-15) para asegurar que cada elemento de `M` es menor que el siguiente. Esta función también establece implícitamente una cota inferior para las variables de `M`, ya que su primer elemento toma el valor 0. Además, llamando a `order` con `M ++ [trunc(2^(N-1))]`, también se establece una cota superior para dichas variables.
- La línea 7 utiliza la función extra `gen_difs` (líneas 16-18) para generar la lista de variables D que representan las diferencias entre cada par de variables de `M`. Esta función computa la lista `[[int]] Ds` = $[[D_{1-0}, \ldots, D_{(N-1)-0}], [D_{2-1}, \ldots, D_{(N-1)-1}], \ldots, [D_{(N-1)-(N-2)}]]$. Cada D_{i-j} se genera implícitamente como el resultado de la resta $M_j - M_i$. Como tanto M_i como M_j tienen ya un dominio asociado, `gen_difs` sirve también para la inicialización del dominio de las nuevas variables generadas.
- La línea 8 convierte `Ds` en la lista `[int] D` = $[D_{1-0}, \ldots, D_{(N-1)-0}, D_{2-1}, \ldots, D_{(N-1)-1}, \ldots, D_{(N-1)-(N-2)}]$.
- La línea 9 asegura que las variables de `D` toman valores diferentes.
- La línea 10 utiliza la función extra `lbound` (líneas 19 a 20) para imponer una cota inferior para cada D_{i-j} (teniendo en cuenta que, como todas las distancias son distintas, su suma debe ser, al menos, la suma de los primeros $j - i$ números naturales). El segundo argumento de `lbound` utiliza otra función extra `sums_nats` (líneas 21-22). Esta computa la lista infinita $[s_1, s_2 \ldots]$, donde cada s_i representa la suma de los primeros i números naturales.

  La regla de `lbound` aplica `zipWith (#<=)` $[s_1, s_2 \ldots]$ a cada elemento de `Ds`, computando bajo demanda $[s_1, s_2 \ldots]$ y compartiéndola para los diferentes elementos de `Ds`. Tomando como ejemplo el primer elemento de `Ds`, el cómputo de `zipWith (#<=)` $[s_1, s_2, s_{n-1}]$ $[D_{1-0}, \ldots, D_{(N-1)-0}]$ impone que $D_{1-0} >= s_1$, $\ldots$, $D_{(N-1)-0} >= s_{n-1}$. La aplicación de la función de orden superior `map` asegura que `zipWith` se aplica a cada elemento de `Ds`. Como las restricciones relacionales soportan reificación [132], el no asignar el resultado de estas a una variable Booleana se puede entender como un azúcar sintáctico en el que solo se imponga la versión de la restricción que haga `true` el resultado devuelto. Por lo tanto, la lista resultante de la aplicación de `(map (zipWith (#<=) sum_nats) Ds)` es `[[bool]]` (con todos los elementos de la lista siendo `true`). A continuación, `foldl (++) []` convierte `[[bool]]` en `[bool]`. Finalmente, `foldl (/\) true` toma la lista `[bool]`, devolviendo `true` como resultado (asegurando así la restricción de igualdad de la línea 10).

- La línea 11 impone que la primera variable de D sea menor que la última.
- Finalmente, la línea 12 utiliza la primitiva $\mathcal{FD}$ `labeling` para especificar una estrategia de búsqueda. El segundo argumento especifica que `M` es la lista de variables a etiquetar (asignar valores). El primer argumento especifica una función de coste para la búsqueda, donde la última variable de `M` se debe minimizar. Como no se hace explícito ningún criterio de selección de variables o valores del dominio, las variables se etiquetan en su orden textual, y el dominio de cada variable se etiqueta en un orden creciente.

1.4 Contribuciones de la tesis

Con la gran variedad de paradigmas disponibles, actualmente la comunidad CP($\mathcal{FD}$) es grande y muy productiva, desarrollando una gran cantidad de sistemas y aplicaciones. Existen numerosas conferencias y revistas que incluyen a CP($\mathcal{FD}$) entre sus temas a tratar, presentando múltiples aplicaciones industriales. Sin embargo, mientras que estas aplicaciones se encuentran más o menos repartidas entre las que siguen enfoques CP($\mathcal{FD}$) algebraicos, C++ CP($\mathcal{FD}$) y CLP($\mathcal{FD}$), parece que CFLP($\mathcal{FD}$) no ha atraído la atención de la comunidad CP($\mathcal{FD}$). Por ejemplo, en los dos últimos años, las ediciones de estas conferencias y revistas incluyen múltiples aplicaciones de los sistemas MiniZinc, Gecode y SICStus Prolog, pero no incluyen ninguna aplicación basada en un enfoque CFLP($\mathcal{FD}$).

Dada esta falta de aplicaciones reales de CFLP($\mathcal{FD}$), esta tesis se centra en una análisis empírico sobre la aplicabilidad de CFLP($\mathcal{FD}$) para abordar a un CSP o COP presente en la industria. La investigación realizada se centra en el sistema CFLP($\mathcal{FD}$) $\mathcal{TOY}(\mathcal{FD})$, implementado en SICStus Prolog, y con la capacidad para resolver igualdades y desigualdades sintácticas (mediante un resolutor de Herbrand: $\mathcal{H}$), así como restricciones $\mathcal{FD}$ (mediante un resolutor CP($\mathcal{FD}$)). Su lenguaje de modelado y su sobrecarga a la hora de resolver los problemas se analiza en detalle. Estos resultados de modelado y resolución se comparan con los de otros sistemas CP($\mathcal{FD}$) de vanguardia, tanto algebraicos como C++ CP($\mathcal{FD}$) y CLP($\mathcal{FD}$). Para acometer todos y cada uno de estos objetivos, la investigación se ha dividido en tres partes que se describen a continuación.

La primera parte de la investigación tiene por objeto la mejora del rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD})$. El sistema del que se parte utiliza el resolutor subyacente de SICStus `clpfd` (por lo que el sistema se denota como $\mathcal{TOY}(\mathcal{FD}s)$). En esta tesis se integran en $\mathcal{TOY}(\mathcal{FD})$ los resolutores C++ CP($\mathcal{FD}$) de vanguardia Gecode e ILOG Solver, dando lugar respectivamente a las nuevas versiones del sistema $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$. Además, para estas nuevas versiones se aumenta la capacidad expresiva del lenguaje $\mathcal{TOY}(\mathcal{FD})$, añadiendo estrategias de búsqueda *ad hoc*. Las principales contribuciones han sido las siguientes:

- Desarrollo de un esquema para la integración de resolutores C++ CP($\mathcal{FD}$) en $\mathcal{TOY}(\mathcal{FD})$, en un contexto fácilmente aplicable a otros sistemas CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$) implementados en Prolog. Este esquema ha resultado ser ciertamente génerico, puesto que ha permitido integrar dos resolutores diferentes, como son Gecode e ILOG Solver, siguiendo los pasos que se describen en el esquema.
  - Identificación de los diferentes procedimientos requeridos por $\mathcal{TOY}(\mathcal{FD})$ para coordinar un resolutor CP($\mathcal{FD}$), creando una interfaz abstracta y extensible entre el sistema y resolutor (que incluye código de pegamento entre sus componentes Prolog y C++).
  - Gestión de la falta de correspondencia entre las variables, restricciones y tipos soportados por el sistema y resolutor.
  - Adaptación del resolutor C++ CP($\mathcal{FD}$) para cumplir con los requisitos de un sistema CFLP($\mathcal{FD}$), tales como el razonamiento por modelos, el uso de múltiples estrategias de búsqueda (intercaladas con el almacenamiento de nuevas restricciones) y el uso de un modo de propagación tanto incremental como por lotes.
- Aumento de la capacidad expresiva del lenguaje en las versiones del sistema $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$ (ya que son estas las que proporcionan una mayor capacidad de resolución). En concreto se soportan ocho nuevas primitivas de búsqueda parametrizables que proporcionan una especificación de búsqueda más detallada al resolutor (en un contexto fácilmente aplicable a otros sistemas CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$) implementados en Prolog e integrando resolutores C++ CP($\mathcal{FD}$) externos).
  - Estas primitivas incluyen conceptos novedosos de búsqueda (soportando incluso búsquedas incompletas) que no están directamente disponibles en las bibliotecas de Gecode ni de ILOG Solver. En concreto soportan una exploración exhaustiva en anchura sobre los primeros niveles del árbol (ordenando posteriormente los nodos satisfactibles según un criterio específico). También soportan la fragmentación del dominio de las variables, acotando cada una de ellas a un subconjunto de los valores de su dominio (en lugar de asignándolas directamente a un valor concreto). Por último, tanto las estrategias de asignación como de fragmentación se pueden aplicar únicamente a un subconjunto de las variables involucradas.

    Por otra parte, algunos de los criterios de búsqueda de estas primitivas se pueden especificar directamente en el lenguaje $\mathcal{TOY}(\mathcal{FD})$. Además, el propio $\mathcal{TOY}(\mathcal{FD})$ permite combinar fácilmente varias primitivas (para construir nuevas estrategias de búsqueda complejas), así como hacer uso del razonamiento por modelos para aplicar diferentes escenarios de búsqueda a la resolución de un problema.

- Análisis del rendimiento de las tres versiones de $\mathcal{TOY}(\mathcal{FD})$, usando para ello un conjunto mímimo de problemas de prueba, formado por tres CSP clásicos (series mágicas, reinas y números de Langford) y por un COP clásico (reglas de Golomb). Los resultados revelan que $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$ mejoran claramente el rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD}s)$. Por otra parte, las nuevas primitivas de búsqueda permiten mejorar aún más el rendimiento de $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$.
  - El conjunto de problemas de prueba es suficientemente completo, ya que abarca todo el repertorio de restricciones $\mathcal{FD}$ soportadas por $\mathcal{TOY}(\mathcal{FD})$. Además, mediante el uso de diferentes instancias por problema, se analiza el rendimiento de $\mathcal{TOY}(\mathcal{FD})$ a medida que la complejidad del problema aumenta. Finalmente, se analizan las primitivas de búsqueda a aplicar para cada problema, basándose en la estructura de las soluciones de cada uno de estos.

La segunda parte de la investigación presenta dos aplicaciones reales de $\mathcal{TOY}(\mathcal{FD})$. En primer lugar, se presenta un problema de *asignación de trabajadores a turnos de trabajo* (del inglés *Employee Timetabling Problem: ETP*), proveniente de la industria de las comunicaciones. En segundo lugar, se presenta un análisis empírico de la complejidad del problema de *asignación de elementos unidimensionales a contenedores* (del inglés *one-dimensional Bin Packing Problem: BPP*), necesario para resolver intancias generalizadas provenientes, en particular, de la industria de los centros de datos. Las principales contribuciones han sido las siguientes:

- Descripción de un algoritmo no monolítico para la especificación de una versión genérica del ETP, donde los trabajadores están divididos en diferentes equipos. Comparativa entre el rendimiento de resolución alcanzado y el que se obtuvo anteriormente para el conjunto de problemas de prueba.
  - La compleja formulación del problema explota la alta expresividad de $\mathcal{TOY}(\mathcal{FD})$. Esta es totalmente paramétrica en el número de días del calendario, el número de equipos (y el número de trabajadores por equipo), la periodicidad con que el trabajador adicional se puede seleccionar (y el factor extra al que sus horas de trabajo se deben pagar), el número de diferentes tipos de jornadas de trabajo (y los turnos concretos solicitados en cada una de ellas), las ausencias de trabajadores y el nivel de homogeneidad requerido en la distribución de cada tipo de turno de trabajo entre los trabajadores de cada equipo.
  - El enfoque utilizado para resolver el problema divide el espacio de búsqueda inicial en tantos subespacios como asignaciones posibles de equipos a días existan, explorando solo aquellos que sean satisfactibles. Para cada uno de estos, se descompone nuevamente al problema; en este caso en

tantos subproblemas independientes como equipos haya (siendo la complejidad computacional del subproblema asociado a cada equipo exponencialmente menor que la del subespacio inicial).

  - Los resultados de rendimiento (tanto en términos de la version concreta de $\mathcal{TOY}(\mathcal{FD})$ utilizada, como en la aplicación de estrategias de búsqueda *ad hoc*) son similares a los obtenidos para el conjunto de problemas clásicos de prueba, aunque las diferencias obtenidas entre las diferentes instancias resueltas son ahora mayores.

- Resolución de un conjunto de instancias del problema BPP (generadas paramétricamente mediante la conocida distribución de Weibull [201]), aplicando para dicha resolución dos modelos CP($\mathcal{FD}$) equivalentes (Gecode y $\mathcal{TOY}(\mathcal{FD}g)$) y cuatro heurísticas.

  - Weibull permite generar una gran variedad de distribuciones (en lo que al tamaño de los elementos se refiere), ya que su alta flexibilidad permite representar a casi cualquier distribución unimodal. Su modelo paramétrico se usa para representar (de manera muy precisa) a instancias reales del problema BPP. En concreto se utilizan técnicas de *máxima verosimilitud* y *gráficos Q-Q* para observar la calidad del ajuste. Además, se utilizan los tests estadísticos de Kolmogorov-Smirnov y χ^2 para probar de manera más rigurosa dicho ajuste.
  - El conjunto de instancias se construye utilizando 199 combinaciones diferentes de los parámetros de Weibull (generando 100 instancias por cada combinación). Además, se proponen once escenarios diferentes, que varían el tamaño del contenedor al asignar a este un factor multiplicador (de entre 1,0 y 2,0, con incrementos de 0,1) del tamaño del elemento más grande de la distribución. Finalmente, se utilizan archivos de procesamiento por lotes para establecer sesiones de resolución del conjunto de instancias, tanto para los sistemas CP($\mathcal{FD}$) como para las heurísticas.
  - Los resultados obtenidos revelan que, tanto CP($\mathcal{FD}$) como las heurísticas, son adecuados para resolver el problema BPP. Dependiendo de la instancia concreta a resolver (combinación de parámetros de Weibull) y el tamaño del contenedor (escenario elegido), ambas técnicas proporcionan un compromiso entre el tiempo empleado para resolver el problema y la calidad de la solución obtenida.

La tercera parte de la investigación posiciona a $\mathcal{TOY}(\mathcal{FD})$ con respecto a sistemas CP($\mathcal{FD}$) de vanguardia. En concreto con los sistemas CP($\mathcal{FD}$) algebraicos MiniZinc e ILOG OPL, los sistemas C++ CP($\mathcal{FD}$) Gecode e ILOG Solver, los sistemas CLP($\mathcal{FD}$) SICStus Prolog y SWI-Prolog, y el sistema CFLP($\mathcal{FD}$) PAKCS. Los resultados fomentan el

uso de $\mathcal{TOY}(\mathcal{FD})$ (y del propio paradigma CFLP($\mathcal{FD}$)), demostrando que este es competitivo con respecto a cualquiera de los otros sistemas para el modelado y resolución de los dos COP propuestos (el puzzle clásico de Golomb y el problema real ETP presentado previamente). Las principales contribuciones han sido las siguientes:

- Desarrollo de una comparativa para modelar dos COP entre los sistemas MiniZinc, ILOG OPL, Gecode, ILOG Solver, SICStus Prolog, SWI-Prolog, PAKCS y $\mathcal{TOY}(\mathcal{FD})$.
  - Debido a la simplicidad de la formulación del problema de Golomb, este proporciona una visión general sobre conceptos básicos del modelado de un COP, como son la abstracción del resolutor de restricciones, la especificación de las variables $\mathcal{FD}$, restricciones $\mathcal{FD}$ y estrategia de búsqueda, así como la visualización de las soluciones encontradas. En cuanto al ETP, su compleja formulación (totalmente paramétrica, no monolítica y con componentes independientes de CP($\mathcal{FD}$)), explota la capacidad expresiva de los diferentes paradigmas, permitiendo analizar en detalle las ventajas e inconvenientes de cada uno de ellos.
  - La comparativa incluye varios ejemplos de código, para poner en contexto las ideas que se presentan. Además, se proporciona el código completo de cada modelo (para cada problema y sistema). La comparativa de expresividad también incluye la cantidad de líneas de código utilizas en cada caso.
- Desarrollo de una comparativa para resolver dos COP entre los sistemas MiniZinc, ILOG OPL, Gecode, ILOG Solver, SICStus Prolog, SWI-Prolog, PAKCS, $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ y $\mathcal{TOY}(\mathcal{FD}s)$.
  - Establecer un marco común para la ejecución de los experimentos, teniendo en cuenta las versiones de los sistemas, las restricciones globales que se utilizan (y sus algoritmos de filtrado) y la medición del tiempo empleado. Reutilizar las tres instancias por problema (con tiempos de resolución de décimas de segundo, segundos y minutos, respectivamente) que se consideraron en los análisis anteriores.
  - Comparar el rendimiento de resolución de los diez sistemas, analizando su *ranking* y sobrecarga con respecto al sistema más rápido. Analizar el orden de rendimiento existente entre los sistemas que utilizan un resolutor de restricciones común, especializando posteriormente la comparativa para los sistemas que utilicen el resolutor de Gecode, ILOG Solver y SICStus `clpfd` (dedicando un análisis aislado de cada uno de ellos).
  - Comparar el rendimiento de $\mathcal{TOY}(\mathcal{FD}g)$ con el del modelo nativo Gecode (asimismo el de $\mathcal{TOY}(\mathcal{FD}i)$ con el del modelo nativo ILOG Solver, y el de $\mathcal{TOY}(\mathcal{FD}s)$ con el del modelo nativo SICStus Prolog, respectivamente), para analizar y justificar la sobrecarga de cada versión de $\mathcal{TOY}(\mathcal{FD})$.

2 Mejora del rendimiento de $\mathcal{TOY}(\mathcal{FD})$

Debido a la naturaleza combinatoria de los CSP y COP que se abordan con $\mathcal{TOY}(\mathcal{FD})$ se espera que, a medida que las instancias aumenten lo suficiente, la mayor parte del tiempo de resolución de un problema se emplee en la búsqueda. En este contexto, y siguiendo la ley de Amdahl (que establece que la mejora de rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que utiliza dicho componente) parece claro que hay dos tipos de aproximaciones adecuadas para aumentar la eficiencia de resolución de $\mathcal{TOY}(\mathcal{FD})$. La primera consiste en reemplazar al resolutor de restricciones subyacente al sistema por nuevos resolutores externos con mejor rendimiento. Es decir, manteniendo la misma formulación del problema en $\mathcal{TOY}(\mathcal{FD})$ (específicamente, la misma red de restricciones y la misma estrategia de búsqueda), utilizar nuevos resolutores de restricciones capaces de efectuar el proceso de búsqueda más rápido. La segunda alternativa consiste en sustituir la estrategia de búsqueda original por una nueva estrategia *ad hoc*, que explote el conocimiento acerca de la estructura del problema y sus soluciones. Es decir, manteniendo el mismo resolutor para acometer la búsqueda, modificar el modelo $\mathcal{TOY}(\mathcal{FD})$ para especificar una nueva estrategia de búsqueda que requiera una menor exploración para encontrar las soluciones.

Esta sección presenta la implementación de las dos alternativas en $\mathcal{TOY}(\mathcal{FD})$: la sección 2.1 presenta un esquema para integrar a los resolutores externos C++ CP($\mathcal{FD}$) de Gecode e ILOG Solver en $\mathcal{TOY}(\mathcal{FD})$, dando lugar a las nuevas versiones del sistema $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$, respectivamente. La sección 2.2 describe las nuevas primitivas de búsqueda añadidas para estas versiones del sistema que permiten especificar estrategias de búsqueda *ad hoc*.

2.1 Esquema para integrar resolutores C++ CP($\mathcal{FD}$)

$\mathcal{TOY}(\mathcal{FD})$ está implementado en SICStus Prolog. Su arquitectura contiene un resolutor de Herbrand ($\mathcal{H}$), que permite gestionar restricciones de igualdad y desigualdad sintácticas, así como un resolutor $\mathcal{FD}$, que permite gestionar restricciones de dominio finito. Se necesita una interfaz para coordinar al resolutor $\mathcal{FD}$ con la semántica operacional del sistema. Esta interfaz consiste en un conjunto de predicados Prolog $pred_1, \ldots, pred_k$, cada uno de ellos implementando un comando del sistema mediante el uso de la API del resolutor. En el caso de $\mathcal{TOY}(\mathcal{FD}s)$, el sistema utiliza el resolutor CP($\mathcal{FD}$) subyacente de SICStus `clpfd`, dando lugar a una interfaz sencilla y elegante entre el sistema y el resolutor. En concreto las variables de $\mathcal{TOY}$ y `clpfd` tienen una representación común como variables lógicas Prolog. Además, `clpfd` proporciona una gestión de la vuelta atrás transparente al usuario. Este restaura implícitamente el almacén de restricciones (de ahora en adelante $\texttt{store}^{\mathcal{FD}}$) al punto concreto de la

resolución del objetivo al que el sistema ha retornado. Del mismo modo, `clpfd` proporciona una gestión implícita del uso de múltiples exploraciones de búsquedas intercaladas con la imposición de nuevas restricciones (donde cada búsqueda actúa sobre su red de restricciones asociada). Finalmente, `clpfd` utiliza implícitamente un modo de propagación incremental (realizando una propagación de restricciones cada vez que una nueva restricción se impone sobre el almacén [197]).

La integración en $\mathcal{TOY}(\mathcal{FD})$ de nuevos resolutores C++ CP($\mathcal{FD}$) aumenta la complejidad de la interfaz. La primera dificultad resulta de comunicar sistema con resolutor (puesto que ahora está conectando a componentes implementados en lenguajes diferentes, como son Prolog y C++). Además, las variables, restricciones y tipos difieren entre ambos componentes, lo que provoca un desajuste de impedancia. Tres dificultades adicionales provienen de la adaptación de un resolutor C++ CP($\mathcal{FD}$) a los requisitos de un sistema CFLP($\mathcal{FD}$). En primer lugar, en estos resolutores el razonamiento por modelos no está soportado, pues es una característica asociada a la programación lógica. Por lo tanto, la API del resolutor no proporciona un método para eliminar a una restricción del almacén. En segundo lugar, en estos resolutores se espera una única exploración de búsqueda que se aplicará sobre la red de restricciones completa. Para utilizar varias exploraciones de búsquedas intercaladas con la imposición de nuevas restricciones es necesario utilizar (y coordinar) a varios resolutores. En tercer lugar, en estos resolutores la propagación incremental puede no estar soportada, ya que se espera que la primera vez que se propague a la red completa de restricciones sea justo al iniciarse la exploración de búsqueda.

Las siguientes subsecciones presentan un esquema genérico para afrontar cada una de estas dificultades.

2.1.1 Comunicación

Los predicados Prolog $pred_1, \ldots, pred_k$, que actúan como interfaz con la API del resolutor, requieren ahora acceder a código C++. SICStus proporciona un marco de comunicación Prolog-C++ que permite definir un predicado prototipo en Prolog pero cuya implementación está en realidad contenida en una función C++. Estos prototipos incluye el número de argumentos, especificando el modo de cada uno de ellos (entrada o salida), así como su tipo. SICStus proporciona una conversión entre los parámetros Prolog y C++, que incluye una representación C++ para los términos Prolog.

Para integrar resolutores C++ CP($\mathcal{FD}$), la funcionalidad de los procedimientos $\mathcal{TOY}(\mathcal{FD})$ se implementa mediante un conjunto de funciones C++ $f_1, \ldots, f_n$, que acceden la API del resolutor. Por ejemplo, f_i podría estar encargada de crear una nueva variable $\mathcal{FD}$, f_j de imponer una restricción `domain` y f_z de acometer la propagación de restricciones del almacén. Estas funciones C++ $f_1, \ldots, f_n$ son accesibles respectivamente desde Prolog mediante los predicados prototipo $p_1, \ldots, p_n$. Así, cualquier

predicado $pred_i$ de la interfaz gestiona el comando impuesto por el sistema utilizando p_i, p_j y p_z tantas veces como sea necesario.

Por otra parte, la interfaz se debe extender con estructuras de datos Prolog y C++ que sean accesibles a los predicados Prolog y que actúen como pegamento para la integración del resolutor. Mientras que las estructuras de datos C++ se almacenan como variables globales, las estructuras Prolog se almacenan en el término `fd_glue`. Para ello, `store`$^{\mathcal{H}}$ se reemplaza por el par `(store`$^{\mathcal{H}}$`, fd_glue)`, donde el segundo componente representa a las estructuras de datos Prolog adicionales.

2.1.2 Representación

Cada resolutor C++ CP($\mathcal{FD}$) proporciona su propia representación para variables y restricciones $\mathcal{FD}$, diferente de las de $\mathcal{TOY}(\mathcal{FD})$. Por lo tanto, en la implementación, cada variable y restricción contiene dos representaciones diferentes (aunque equivalentes), y la interfaz proporciona la conexión entre ambas. Para cada nueva restricción pC$_i$ impuesta por el objetivo $\mathcal{TOY}(\mathcal{FD})$, la interfaz genera una restricción equivalente cC$_i$ que se impone sobre `store`$^{\mathcal{FD}}$ (véase la figura 3). Además, las variables $\mathcal{FD}$ asociadas a dicha restricción se atribuyen explícitamente. Esto permite identificar como restricción $\mathcal{FD}$ a cualquier futura restricción de igualdad o desigualdad sintáctica que involucre alguna de estas variables.

La comunicación entre las dos representaciones de cada variable es bidireccional. Por un lado, cuando se gestiona pC$_i$, sus variables lógicas asociadas pV$_1$ ... pV$_k$ ser deben conectar con las variables de decisión equivalentes cV$_1$ ... cV$_k$, para que la restricción equivalente cC$_i$ se imponga sobre ellas. Sin embargo, ni la API de Prolog ni el de `store`$^{\mathcal{FD}}$ proporcionan métodos para la obtención de sus variables almacenadas. Por ello, se deben utilizar los vectores auxiliares pV y cV, que contienen en su posición *i*-ésima a las representaciones lógica y de decisión (respectivamente) de la *i*-ésima variable involucrada en la resolución del objetivo. Por otra parte, una solución de un objetivo $\mathcal{TOY}(\mathcal{FD})$ debe mostrar los dominios obtenidos para las variables, así como todas las restricciones del almacén que no hayan sido simplificadas hasta convertirse en trivialmente ciertas. En primer lugar, esto implica que las variables de cV que se han asignado a un valor mediante la propagación de restricciones deben disparar la unificación a dichos valores de sus variables lógicas equivalentes de pV. Para ello, se define una nueva clase de restricción unaria d que impone una restricción a `store`$^{\mathcal{FD}}$ por cada variable de decisión cV$_i$ creada. Esta restricción se propaga cuando el dominio de la variable involucrada se acota al valor k, almacenando el par `(i, k)` en la estructura de datos C++ cV_bound. Esta estructura se vacía antes de la propagación de restricciones para que, tras esta, la estructura se pueda recorrer indentificando todas las variables de cV que se han asignado a un valor. Con esta información se genera un término Prolog que la interfaz utiliza para unificar a las variables pV equivalentes. En

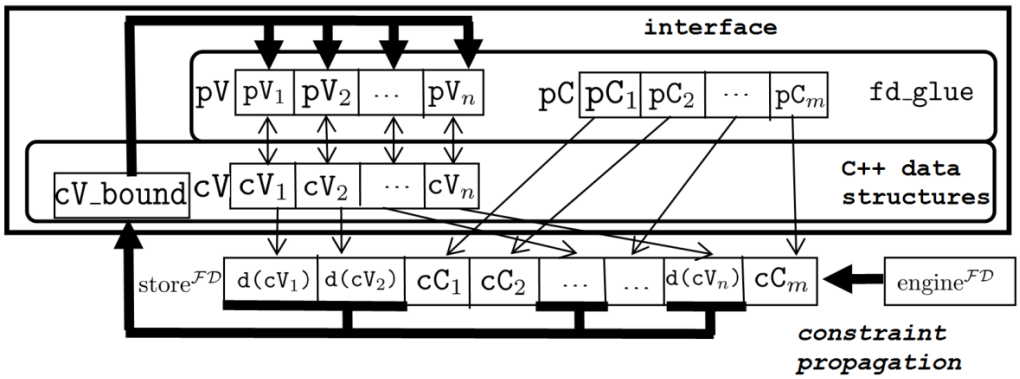

Figura 3. Estructuras de Datos

segundo lugar, dada una variable como argumento, la API del resolutor proporciona un método para mostrar su dominio. Por lo tanto, se recorre cV solicitando el dominio de cada variable cV_i sin asignar. Por último, mientras que `clpfd` sí que proporciona un método para acceder a $\texttt{store}^{\mathcal{FD}}$, un resolutor C++ CP($\mathcal{FD}$) no, ya que estos esperan que todas las variables hayan sido asignadas a valores. Por lo tanto, para mostrar el contenido de $\texttt{store}^{\mathcal{FD}}$, cada restricción pC_i procesada se almacena explícitamente en una lista Prolog pC. Respecto a la consistencia de pV y cV, se recorre la primera, mostrando en la solución únicamente las restricciones que no han sido simplificadas hasta convertirse en trivialmente ciertas.

2.1.3 Vuelta atrás

El razonamiento por modelos ofrece posibilidades interesantes para la formulación de un CSP o COP (véase [134] como ejemplo donde una vuelta atrás cronológica juega un papel clave en la mejora de rendimiento de un problema de asignación de recursos para un entorno académico). $\mathcal{TOY}(\mathcal{FD})$ soporta el razonamiento por modelos mediante el uso de funciones indeterministas, cuyas múltiples reglas se exploran por vuelta atrás, restaurando $\texttt{store}^{\mathcal{H}}$ y $\texttt{store}^{\mathcal{FD}}$. Según la semántica operacional de $\mathcal{TOY}(\mathcal{FD})$, las restricciones de un objetivo se procesan en su orden textual. Por ello, la noción de vuelta atrás consiste en eliminar (de $\texttt{store}^{\mathcal{H}}$ y $\texttt{store}^{\mathcal{FD}}$) las k restricciones primitivas procesadas desde el último punto de elección. Esta vuelta atrás afecta al resolutor $\mathcal{FD}$ si alguna de esas últimas $k1 \in \{1,\ldots,k\}$ restricciones primitivas a eliminar es una restricción $\mathcal{FD}$, incluyendo asimismo a las últimas $k2 \geq 0$ variables $\mathcal{FD}$ creadas (asociadas únicamente a estas k1 restricciones).

Cuando se produce una vuelta atrás, el motor Prolog restaura automáticamente $\texttt{store}^{\mathcal{H}}$ y `fd_glue`, pero no así a $\texttt{store}^{\mathcal{FD}}$ ni a las estructuras de datos C++. La figura 4 presenta un ejemplo donde los elementos de pV y pC que se eliminan automáticamente se muestran en líneas discontinuas. Únicamente los predicados Prolog de la

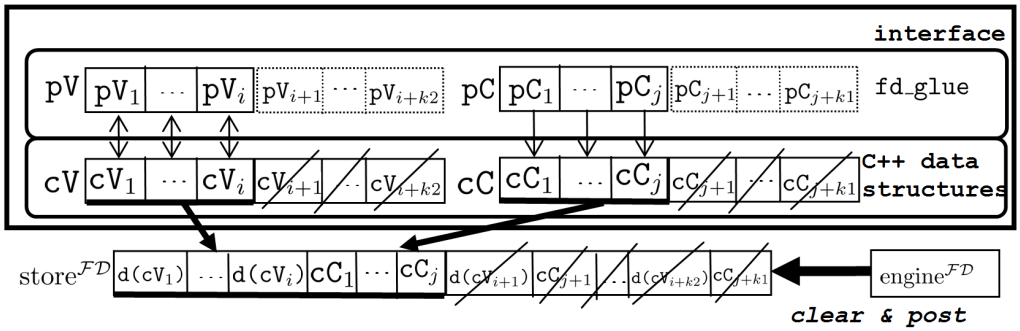

Figura 4. Identificación de vuelta atrás y restauración

interfaz pueden identificar que se ha producido una vuelta atrás, y que este ha afectado al resolutor $\mathcal{FD}$. Para ello, basta con que cada uno de estos predicados compare los tamaños de `store`$^{\mathcal{FD}}$ y pV+pC al iniciar su ejecución. En este contexto, puede darse el caso de que, tras producirse una vuelta atrás y mientras el resolutor $\mathcal{FD}$ aún se mantiene desincronizado, se procesen nuevas restricciones $\mathcal{H}$. Sin embargo, esto no es problemático pues en cuanto se produzca cualquier tipo de interacción con el resolutor $\mathcal{FD}$, el predicado Prolog que gestione esta interacción detectará la vuelta atrás y sincronizará previamente al resolutor.

Para eliminar las restantes k2 variables de cV se comparan los tamaños de pV y cV. Para eliminar las restantes k1+k2 restricciones de `store`$^{\mathcal{FD}}$ primero se vacía completamente el almacén y después se imponen nuevamente `i+j` restricciones (ya que la API del resolutor no proporciona un método para eliminar una única restricción de `store`$^{\mathcal{FD}}$). Mientras que cV se recorre para crear e imponer nuevamente las restricciones d, para imponer a las restricciones primitivas se consideran dos posibilidades: la primera recorre pC, traduciendo de nuevo a cada $\mathtt{pC}_i$ a su restricción equivalente $\mathtt{cC}_i$. La segunda alternativa replica las restricciones primitivas impuestas sobre `store`$^{\mathcal{FD}}$ en el vector cC. Se selecciona esta segunda opción ya que, en general, la gestión de cC es más eficiente que traducir nuevamente a las restricciones $\mathtt{pC}_i$. Para eliminar a las restantes k1 restricciones de cC se compara previamente a los tamaños de pC y cC. Finalmente, se recorre cC para imponer nuevamente las restricciones sobre `store`$^{\mathcal{FD}}$.

2.1.4 Múltiples exploraciones de búsqueda

El uso de múltiples etiquetados o `labelings`, intercalados con la imposición de nuevas restricciones, ofrece posibilidades interesantes para la formulación de un CSP o COP (véase [207] como ejemplo de un problema de asignación de recursos para un entorno académico que precisa de dos etapas, cada una de ellas con una exploración de búsqueda asociada a una red de restricciones concreta). En $\mathcal{TOY}(\mathcal{FD})$, esta característica está soportada (un `labeling` se considera como una simple expresión) pero,

en la mayor parte de los resolutores C++ CP($\mathcal{FD}$), el método de la API para imponer nuevas restricciones sobre $\texttt{store}^{\mathcal{FD}}$ no es aplicable cuando el resolutor se encuentra en medio de una exploración de búsqueda. Para solventar esta dificultad, $\texttt{store}^{\mathcal{FD}}$ se dedica únicamente para la imposición de restricciones, y el vector de resolutores auxiliares ss para gestionar los labelings $\texttt{l}_1 \ \ldots \ \texttt{l}_l$ que aparezcan durante la resolución del objetivo (véase la figura 5). Por lo tanto, el resolutor principal (o $\texttt{engine}^{\mathcal{FD}}$) nunca ejecuta un labeling y así, al no encontrarse nunca en modo búsqueda, el uso de múltiples labelings puede intercalarse con la imposición de nuevas restricciones mediante una sincronización entre los almacenes de $\texttt{store}^{\mathcal{FD}}$ y cada $\texttt{store\_ss}_i$:

- Al crear un nuevo resolutor $\texttt{ss}_i$ se recorre cC, imponiendo las restricciones sobre $\texttt{store\_ss}_i$. En este caso no se deben imponer las restricciones d ya que, durante la búsqueda, las ramas que conduzcan a fallos asignarán valores erróneos a las variables. Por lo tanto, al inicio de la búsqueda, $\texttt{store\_ss}_i$ es consistente con el estado actual de la resolución del objetivo, y $\texttt{l}_i$ se asigna al resolutor *i*-ésimo (o $\texttt{engine\_ss}_i$), que acometerá la búsqueda de soluciones.
- Como el labeling es un proceso de enumeración, cuando $\texttt{engine\_ss}_i$ encuentra una rama solución, esta surge de imponer un conjunto de restricciones de igualdad sobre $\texttt{store\_ss}_i$, y dicha rama puede representarse como un conjunto de pares (variable, valor). Este rama solución (o conjunto de restricciones) se debe imponer sobre cC y $\texttt{store}^{\mathcal{FD}}$ para que estos sean consistentes con el efecto de haber realizado la búsqueda y poder así continuar la resolución del objetivo. Para la sincronización se recorre cV, solicitando a $\texttt{engine\_ss}_i$ el dominio de cada variable $\texttt{cV}_i$, e imponiendo la restricción de igualdad si dicha variable se ha asignado a un valor durante la búsqueda.

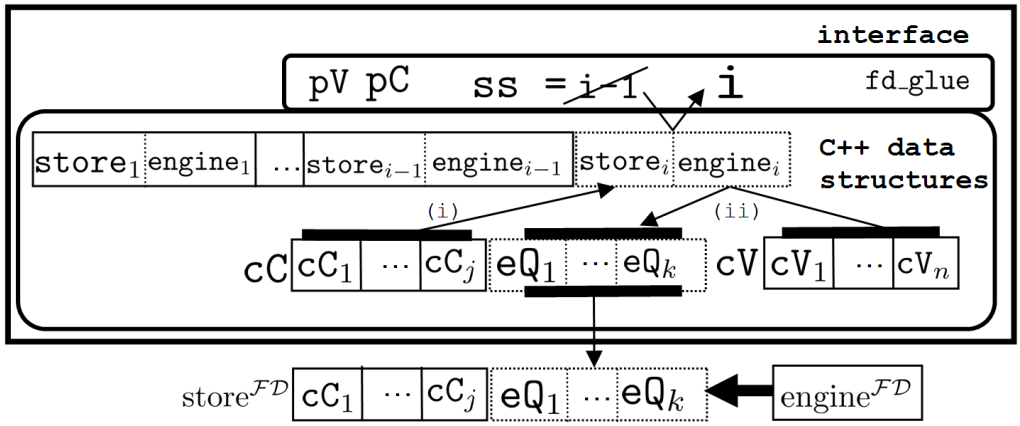

Figura 5. Gestión del etiquetado

Con respecto a la interfaz, el predicado Prolog dedicado a gestionar las expresiones `labeling` toma a $\texttt{l}_i$ como argumento, y usa un bucle `repeat` para solicitar a $\texttt{engine\_ss}_i$ que busque las soluciones una a una. Para distinguir la primera llamada a $\texttt{l}_i$ (que solicita la creación de $\texttt{ss}_i$) de las siguientes (que solicitan una nueva solución via vuelta atrás) se guarda el tamaño de `ss` en `fd_glue`, por lo que este se restaura automáticamente cuando se produce la vuelta atrás. En cada iteración del bucle, el predicado Prolog compara ambos tamaños. Cuando no se encuentran más soluciones, se elimina a $\texttt{ss}_i$ y el bucle completo falla.

2.1.5 Propagación incremental y por lotes

El modo de propagación incremental es habitual en los sistemas que soportan un razonamiento por modelos, ya que les permite detectar fallos a lo largo de la resolución del objetivo tan pronto como sea posible. Sin embargo, el modo de propagación por lotes (donde el resolutor realiza la propagación de restricciones bajo demanda) es más habitual en los sistemas C++ CP($\mathcal{FD}$). Mientras que este modo puede mejorar el rendimiento de resolución gracias a propagar un menor número de veces, también es posible que la evaluación de un objetivo que no conduce a ninguna solución continúe hasta que se demande explícitamente una propagación [197]. Se definen las primitivas $\mathcal{TOY}(\mathcal{FD})$ `batch_on` y `batch_off`, que habilitan y deshabilitan (respectivamente) el modo de propagación por lotes. Ambas primitivas se pueden utilizar libremente en los programas $\mathcal{TOY}(\mathcal{FD})$, permitiendo a los usuarios decidir qué partes son propagadas de forma incremental y cuáles por lotes.

Al integrar resolutores C++ CP($\mathcal{FD}$), el modo por lotes implica que las restricciones sean impuestas a `cC`, pero no a $\texttt{store}^{\mathcal{FD}}$. Además, no se crea ni impone ninguna restricción $\texttt{d(cV}_i\texttt{)}$ por cada nueva variable $\texttt{cV}_i$ creada. Cuando el modo incremental vuelve a activarse, las restricciones restantes se deben imponer sobre $\texttt{store}^{\mathcal{FD}}$ antes de realizar la propagación. Por ello, la tupla de enteros (b, bcV, bcC) se añade a `fd_glue`. Mientras que b es un valor binario que representa el modo de propagación, bcV y bcC representan el número de restricciones d y cC (respectivamente) impuestas sobre $\texttt{store}^{\mathcal{FD}}$. En primer lugar, cuando se procesa una primitiva `batch_on`, b toma el valor 1, y bcV y bcC toman respectivamente el valor de los tamaños actuales de cV y cC. En segundo lugar, cuando se procesa la primitiva `batch_off`, b toma el valor 0, y tanto bcV como bcC se comparan respectivamente con los tamaños de cV y cC, para imponer sobre $\texttt{store}^{\mathcal{FD}}$ las restricciones restantes. En tercer lugar, cuando se produce una vuelta atrás a un punto del cómputo en el que b ya tomaba el valor 1, se utilizan los tamaños restaurados de bcV y bcC para imponer sobre $\texttt{store}^{\mathcal{FD}}$ únicamente las restricciones que ya habían sido impuestas previamente a que la primitiva `batch_on` fuera procesada.

El modo de propagación por lotes también se ha implementado en $\mathcal{TOY}(\mathcal{FD}s)$ mediante el uso de los objetivos congelados soportados por SICStus. Se modifica a

store$^{\mathcal{H}}$ para que sea el par (store$^{\mathcal{H}}$, B), donde B representa el modo de propagación utilizado en cada momento. Cuando se procesa una primitiva batch_on, se utiliza a una nueva variable lógica en B, congelando la imposición de toda nueva restricción hasta que B se unifique a un valor. Cuando se procesa una primitiva batch_off, la variable se unifica a B, lo que dispara automáticamente la imposición de todas las restricciones congeladas sobre store$^{\mathcal{FD}}$.

2.2 Estrategias de búsqueda *ad hoc*

En esta sección se presentan ocho nuevas primitivas de búsqueda para especificar estrategias *ad hoc* en $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$, permitiendo al usuario una mayor interacción con el resolutor en la búsqueda de soluciones. Se describen las primitivas y sus componentes (incluyendo su declaración de tipo) desde un punto de vista abstracto (independiente del resolutor). Se presta especial atención a los conceptos novedosos que estas primitivas proporcionan, los cuales no están disponibles en las estrategias de búsqueda predefinidas de Gecode ni de ILOG Solver. Finalmente, también se muestran las posibilidades que $\mathcal{TOY}(\mathcal{FD})$ ofrece para aplicar diferentes estrategias de búsqueda para la resolución de un problema.

2.2.1 Primitivas de asignación

En esta sección se describen cuatro primitivas de búsqueda: lab, que representa una variación de la versión clásica de labeling, incluyendo nuevos criterios de selección de variables y valores, así como la posibilidad de etiquetar solo un subconjunto de las variables involucradas. La segunda, labB, representa una variación de lab, donde únicamente se explora una rama del árbol de búsqueda. La tercera, labW, realiza una exploración exhaustiva en amplitud de los primeros niveles del árbol de búsqueda, ordenando posteriormente los nodos satisfactibles según un criterio especificado. Finalmente, lab0, representa una variación de la versión de optimización de lab.

Primitiva lab

```
lab :: varOrd -> valOrd -> int -> [int] -> bool
```

Esta primitiva devuelve (una a una) todas las posibles combinaciones de valores que satisfacen el conjunto de restricciones impuestas sobre el almacén. Está parametrizada por cuatro componentes básicos. Los dos primeros representan respectivamente los criterios de selección de variables y valores que se usarán en la exploración. Para expresarlos se han definido en $\mathcal{TOY}$ los tipos enumerados varOrd y valOrd, que abarcan todos los criterios predefinidos disponibles en la documentación de Gecode [173]. Estos también incluyen un último caso (userVar y userVal, respectivamente) en el que el usuario especifica en $\mathcal{TOY}(\mathcal{FD})$ su propio criterio de selección de variable

y de valor. El tercer parámetro N representa el número de variables del conjunto que se etiquetarán. Esto representa un concepto novedoso que no está disponible en las estrategias de búsqueda predefinidas de Gecode ni de ILOG Solver. El cuarto argumento representa el conjunto de variables S. Por lo tanto, la búsqueda etiqueta únicamente las primeras N variables de S seleccionadas por el criterio `varOrd`.

Primitiva labB

```
labB :: varOrd -> valOrd -> int -> [int] -> bool
```

Esta primitiva usa los mismos cuatro componentes básicos que `lab`. Sin embargo, su semántica es diferente, ya que sigue los criterios `varOrd` y `valOrd` para explorar únicamente una rama del árbol de búsqueda, sin permitir que se produzca vuelta atrás.

Primitiva labW

```
labW :: varOrd -> bound -> int -> [int] -> bool
```

Esta primitiva realiza una exploración exhaustiva en amplitud de los primeros niveles del árbol de búsqueda, almacenando en una estructura de datos DS cada nodo satisfactible encontrado. Una vez que estos niveles del árbol han sido completamente explorados, se devuelven (uno a uno) los nodos satisfactibles mediante el uso de un criterio para seleccionar y eliminar el *mejor* nodo de DS. El primer parámetro representa el criterio de selección de variables (en este caso no es necesario un criterio de selección de valores, ya que la búsqueda es exhaustiva, por lo que todos los valores se seleccionarán antes de devolver ninguna solución). El segundo parámetro representa el criterio para seleccionar al *mejor* nodo. Para expresarlo en $\mathcal{TOY}(\mathcal{FD})$ se ha definido el tipo enumerado `ord` que puede especificar al nodo que de lugar a un espacio de búsqueda más pequeño o grande (tanto con respecto a las cardinalidades de las variables pertenecientes a `labW`, como a las cardinalidades de todas las variables del almacén). Una vez más, un último caso (`userBound`) permite al usuario especificar el criterio de selección en $\mathcal{TOY}(\mathcal{FD})$. El tercer parámetro especifica los niveles del árbol a explorar. Finalmente, como siempre, el último parámetro representa el conjunto de variables a etiquetar.

La primitiva `labW` representa un concepto novedoso que no está disponible en las estrategias de búsqueda predefinidas de Gecode ni de ILOG Solver. Sin embargo, debe ser utilizado con cuidado, ya que explorar el árbol en mucha profundidad puede llevar a una explosión en el número de nodos, produciendo problemas de memoria en DS y llegando a ser muy ineficiente (debido al tiempo dedicado a la exploración del árbol y a la selección del *mejor* nodo).

Primitiva labO

```
labO:: optType -> varOrd -> valOrd -> int -> [int] -> bool
```

Esta primitiva realiza una búsqueda estándar de optimización. El primer parámetro `optType` especifica el tipo de optimización a realizar (minimización o maximización) y

la variable a optimizar. Los otros cuatro parámetros son los mismos que en la primitiva `lab`.

2.2.2 Primitivas de fragmentación

```
frag:: domFrag -> varOrd -> intervalOrd -> int -> [int] -> bool
fragB:: domFrag -> varOrd -> intervalOrd -> int -> [int] -> bool
fragW:: domFrag -> varOrd -> bound -> int -> [int] -> bool
fragO:: domFrag->optType->varOrd->intervalOrd->int->[int]->bool
```

Estas cuatro nuevas primitivas están relacionadas con las primitivas `lab*` (donde * especifica a cualquiera de ellas), pero ahora cada variable no será etiquetada (asignada) a un valor, sino *fragmentada* (acotada) a un subconjunto de los valores de su dominio. A modo de ejemplo introductorio se podría pensar en un objetivo que contiene V variables y C restricciones, con `V'` $\equiv$ {V1, V2, V3} siendo un subconjunto de V. La restricción `domain V' 1 9` pertenece a C. Mientras que ninguna restricción de C relaciona a las variables de `V'` entre si, algunas restricciones relacionan a `V'` con el resto de las variables de V.

La figura 6 presenta la exploración del árbol de búsqueda realizada por las primitivas `frag*` y `lab*`, respectivamente. En el caso de `frag*`, las tres variables de `V'` se han fragmentado en los intervalos (1,...,3), (4,...,6) y (7,...,9), dando lugar a 27 nodos hoja, un número exponencialmente menor que el de la exploración de `lab*`: 729. Por un lado, si se supiese que solo existe una solución al problema, entonces las probabilidades de encontrar la combinación correcta de valores para `V'` sería mucho mayor en `frag*` que en `lab*`. Por otro lado, se espera que el espacio de búsqueda restante de los nodos hoja de `lab*` sea exponencialmente menor que el de los nodos de `frag*` debido a la mayor propagación en `V'` (que también se espera que conduzca a una poda mucho mayor en el resto de las variables de V). Por lo tanto, las estrategias de búsqueda `frag*` se pueden entender como una técnica más conservadora donde hay menos expectativas de reducir en gran medida el espacio de búsqueda (ya que las variables no son asignadas) pero donde hay más probabilidades de elegir un subconjunto que contenga valores que conduzcan a soluciones (en lo que puede ser visto como una generalización de *first fail* [94]).

Volviendo a la definición de cada primitiva `frag*`, existen dos diferencias fundamentales con respecto a la definición de su primitiva `lab*` equivalente: En primer lugar, `frag*` contiene como componente básico adicional (primer argumento) el tipo de datos domFrag, que especifica el modo en que se fragmenta el dominio de la variable seleccionada. El usuario puede elegir entre `partition n` e `intervals`. El primero de ellos fragmenta los valores del dominio de la variable en n subconjuntos de la misma cardinalidad. En el caso de `intervals`, este busca intervalos ya existentes en el dominio de las variables, dividiendo el dominio a partir de estos intervalos. Por ejemplo, en el objetivo `domain [X] 0 16, X /= 9, X /= 12`, mientras que

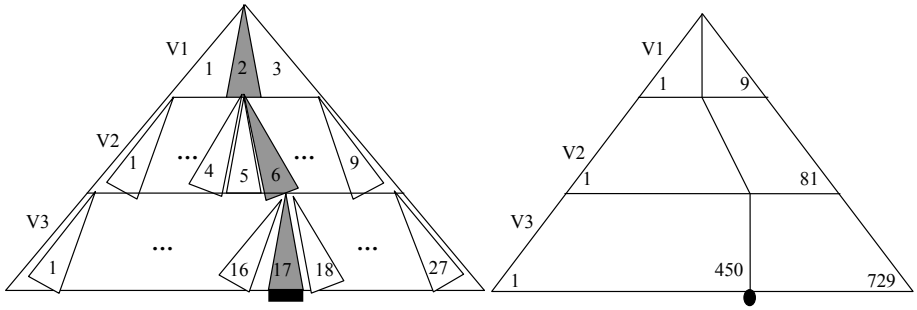

Figura 6. Árbol de búsqueda de `frag` y `lab`

aplicar `partition` 3 a X fragmenta el dominio en los subconjuntos S1 $\equiv$ {0...4}, S2 $\equiv$ {5...8}$\cup${10} y S3 $\equiv$ {11}$\cup${13...16} y aplicar `intervals` fragmenta el dominio en los subconjuntos S1' $\equiv$ {0...8}, S2' $\equiv$ {10...11} y S3' $\equiv$ {13...16}. Como segunda diferencia, `frag*` contiene un tipo de datos enumerado `interval0rd` (que reemplaza al argumento `val0rd` de `lab*`) para especificar el orden en el que se deben seleccionar los diferentes intervalos. Las opciones proporcionadas son: elegir primero el intervalo izquierdo, derecho o central, así como elegir un intervalo aleatoriamente.

Se puede afirmar que las primitivas `frag*` son una herramienta interesante que debe tenerse en cuenta en el contexto de las estrategias de búsqueda, bien como alternativa o como complemento al uso de las primitivas `lab*`. Además, su uso en $\mathcal{TOY}(\mathcal{FD})$ representa un concepto novedoso que no está disponible en las estrategias de búsqueda predefinidas de Gecode ni de ILOG Solver.

2.2.3 Aplicación de diferentes escenarios de búsqueda

$\mathcal{TOY}(\mathcal{FD})$ soporta funciones indeterministas con varias reducciones posibles para los argumentos de una función. Las reglas se aplican siguiendo su orden textual y, tanto los fallos como la solicitud de una nueva solución, disparan la vuelta atrás a la siguiente regla sin explorar. En este contexto, se pueden aplicar secuencialmente diferentes estrategias de búsqueda para resolver un problema. Por ejemplo, siguiendo el ejemplo de la sección anterior, tras imponer V y C al resolutor, el programa y el objetivo $\mathcal{TOY}(\mathcal{FD})$ presentados en la figura 7 utilizan la función indeterminista `f`. Esta especifica tres escenarios diferentes para resolver el problema. Cada escenario termina con un etiquetado exhaustivo del conjunto de variables V. Sin embargo, el espacio de búsqueda s que este etiquetado exhaustivo tiene que explorar se puede reducir notablemente por la evaluación previa de `f`.

Escenario 1: La primera regla de `f` realiza la búsqueda heurística h_1 sobre V' $\equiv$ {V1,V2,V3}. h_1 fragmenta el dominio de V1 en 4 subconjuntos seleccionando uno de ellos al azar. Si la propagación tiene éxito, entonces h_1 asigna V2 y V3 a su valor más pequeño. Si, nuevamente, la propagación tiene éxito (dando lugar a un espacio

de búsqueda s_1), entonces h_1 tiene éxito y el etiquetado exhaustivo explora s_1. Si la propagación falla en uno de esos puntos o el etiquetado exhaustivo no encuentra ninguna solución en s_1, entonces h_1 falla (así como la primera regla de f) ya que tanto las primitivas labB como fragB solo exploran una rama del árbol.

Escenario 2: Se prueba la segunda regla de f que realiza la búsqueda heurística h_2 sobre V'. En esta ocasión, la primitiva fragW se aplica en primer lugar. De este modo, si más adelante bien labB, bien h_2 o bien lab (actuando sobre un espacio de búsqueda s_2) fallaran, entonces se produciría una vuelta atrás sobre fragW, proporcionando el siguiente *mejor* intervalo para V1 (de acuerdo con el criterio de escoger el nodo que dé lugar al subárbol de búsqueda más pequeño). Si después de haber intentado todos los intervalos no se encuentra una solución, entonces h_2 falla (así como la segunda regla de f).

Escenario 3: Si tanto h_1 como h_2 fallan, la tercera regla de f termina trivialmente con éxito. Entonces, el etiquetado exhaustivo se realiza sobre el espacio de búsqueda original obtenido tras imponer V y C sobre el resolutor.

```
f:: [int] -> bool
f [V1,V2,V3] = true <==
  fragB (partition 4) unassignedLeftVar random 0 [V1],
  labB unassignedLeftVar smallestVal 0 [V2,V3]
f [V1,V2,V3] = true <==
  fragW (partition 4) unassignedLeftVar smallestTree 0 [V1],
  labB unassignedLeftVar smallestTotalVars 0 [V2,V3]
f [V1,V2,V3] = true
-------------------------------------------
TOY(FD)> ... (resto del objetivo, que impone la red de
              restricciones C, actuando sobre el
              conjunto de variables V = [V1,V2,V3,...,Vk]),
         f [V1,V2,V3], lab userVar userVal 0 V
```

Figura 7. Aplicación de diferentes estrategias de búsqueda

3 Aplicaciones reales de $\mathcal{TOY}(\mathcal{FD})$

Una vez mejorado el rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD})$, en esta segunda parte de la investigación se presentan dos aplicaciones industriales del sistema. La primera consiste en un problema de asignación de trabajadores a turnos de trabajo (ETP), proveniente de la industria de las comunicaciones. La segunda consiste en un análisis empírico de la complejidad del problema de asignación de elementos unidimensionales a contenedores (BPP), relevante para resolver intancias generalizadas provenientes, en particular, de la industria de los centros de datos. Las secciones 3.1 y 3.2 presentan la aplicación de $\mathcal{TOY}(\mathcal{FD})$ a cada uno de estos problemas.

3.1 ETP

El primer problema industrial que se aborda con $\mathcal{TOY}(\mathcal{FD})$ es ETP. Las siguientes subsecciones proporcionan respectivamente una descripción del problema y del enfoque de resolución utilizado.

3.1.1 Descripción

Un departamento está generando el horario de sus empleados para los próximos nd días. Está organizado con diferentes tipos de jornadas de trabajo $ws \equiv \{ws_1, \ldots, ws_k\}$, cada una de ellas con una cantidad asociada de turnos de trabajo $ws_i \equiv [s_{wsi1}, \ldots, s_{wsim}]$. Así, se identifica a los días $dc \equiv [dc_1, \ldots, dc_{nd}]$ del calendario por el tipo de jornada de trabajo que estos representan (es decir, cada dc_i toma un valor entre $\{ws_1, \ldots, ws_k\}$). Si ws_i proporciona m turnos, entonces para cada día $dc_x = ws_i$ debe haber m trabajadores disponibles en el departamento (ya que cada turno se asigna a un único trabajador). El departamento cuenta con w trabajadores. $nt * ntw$ de ellos son trabajadores fijos, divididos en nt equipos de ntw trabajadores: $\{w_1, \ldots, w_{ntw}\}$ pertenecen al equipo t_1, $\{w_{ntw+1}, \ldots, w_{2*ntw}\}$ pertenecen al equipo t_2, y así sucesivamente. Obviamente, esto incluye también la posibilidad de tener un solo equipo que contiene a todos los trabajadores fijos. En cualquier caso, hay un trabajador extra ew que no pertenece a ningún equipo y solo es seleccionado bajo demanda para hacer frente a las ausencias de los trabajadores fijos. Mientras que ew se considera disponible para los nd días del calendario, los trabajadores fijos pueden estar ausentes algunos días. Esta información se proporciona con $abs \equiv \{(w_{i1}, d_{j1}), \ldots, (w_{il}, d_{jl})\}$, de pares (trabajador fijo, día).

Se selecciona a cada equipo t_i para trabajar cada nt días. Si se selecciona a t_i en el día dc_j, entonces solo se puede seleccionar a $\{w_{(i-1)*ntw+1}, \ldots, w_{i*ntw}\}$ (los trabajadores fijos de ese equipo) y a ew para trabajar en el departamento durante ese día. Al

trabajador extra ew se le puede seleccionar para trabajar solo uno de cada er días consecutivos. Obviamente, esto también incluye la posibilidad de trabajar todos los días, simplemente haciendo que er sea uno. Para cada día dc_j, el equipo t_i seleccionado proporciona (debido a ausencias) $0 \leq a \leq ntw$ trabajadores disponibles para que se les asigne a los m turnos de trabajo del día. Esto da lugar a tres situaciones posibles: en primer lugar, si $a \geq m$, entonces ew y cualquier posible trabajador fijo restante no tiene que trabajar (se asigna a un turno de trabajo de 0 horas). En segundo lugar, si $a = m-1$, entonces ew se selecciona para trabajar, siendo asignado para cubrir uno de los m turnos de dc_j. En tercer lugar, si $a < m-1$, entonces no hay ninguna asignación satisfactible, y se concluye que no se puede seleccionar a t_i para el día dc_j.

Sea $s \equiv \{s_1, \ldots, s_g\}$ el conjunto de los distintos tipos de turnos de trabajo que proporciona ws, se usa T_{t_i,s_z} como una medida de la distribución de los turnos de tipo s_z entre los trabajadores fijos de t_i. Suponiendo que $\{w_{(i-1)*ntw+1}, \ldots, w_{i*ntw}\}$ se asignan a $[cv_1, \ldots, cv_{ntw}]$ turnos de tipo s_z durante los nd días, entonces T_{t_i,s_z} representa la diferencia entre el máximo y el mínimo de estos valores cv. La distribución de estos turnos se ve restringida por T, que representa el máximo valor que puede tomar cualquier $T_{t,s}$. Obviamente, esta restricción se puede evitar haciendo que T tome un valor igual o mayor que $(nd/nt)+1$: cualquier equipo t_i trabaja cada nt días, por lo que trabajará (nd/nt) o $(nd/nt)+1$ días durante el calendario (donde $/$ representa la división entera). Como cada trabajador se asigna a un solo turno de trabajo cada uno de esos días, el valor más alto posible para T_{t_i,s_z} sería aquel en el que un trabajador concreto de t_i se asigna únicamente a turnos del tipo s_z, mientras que otro trabajador del equipo no se asigna a ningún turno de este tipo.

Un calendario contiene un total de h horas de trabajo, por lo que se espera que cada trabajador fijo trabaje $h/(nt*ntw)$ de esas horas. Cualquier hora de más trabajada se considera como una hora extra. La optimización surge en el problema debido a que el departamento debe pagar a los trabajadores fijos por cada hora extra que trabajen, y cualquier hora que trabaja ew se paga como ef horas extras de un trabajador fijo. Obviamente, se puede tratar a ew como a un trabajador fijo simplemente haciendo que ef tome el valor uno. Una asignación óptima en el calendario minimiza el pago de extra horas. En este sentido, es importante señalar que T no pertenece a la función de optimización. El objetivo es minimizar el número de horas extra, no minimizar T. Sin embargo, T representa una medida de la equidad en la asignación de horarios para un calendario, ya que es parametrizable por el usuario. Por ejemplo, dos asignaciones diferentes que impliquen 16 horas extras (una de ellas asignando esas 16 horas a un único trabajador, y la otra dividiendo esas horas entre los trabajadores de un equipo) son equivalentes desde el punto de vista de su optimalidad, pero la segunda es más *justa* con respecto a la asignación del trabajo.

Con esta descripción del problema en mente, una posible instancia consistiría en programar el horario del calendario para la semana que empieza el próximo lunes. Mientras que los días laborables contienen tres turnos de trabajo (de 20, 22 y 24 horas,

respectivamente), los días del fin de semana contienen dos turnos (ambos de 24 horas). El departamento cuenta con 13 trabajadores: 12 de ellos son trabajadores fijos, divididos en 3 equipos de 4 trabajadores. Algunos de estos trabajadores fijos no están disponibles todos los días, con sus ausencias descritas por $abs = [(w_1, d_1), (w_2, d_1), (w_5, d_1), (w_5, d_6), (w_6, d_1), (w_6, d_6), (w_7, d_1), (w_7, d_6), (w_{10}, d_1), (w_{10}, d_6), (w_{11}, d_1), (w_{11}, d_6), (w_{12}, d_1), (w_{12}, d_6)]$. El último es un trabajador extra, que puede ser seleccionado para trabajar un máximo de uno de cada tres días consecutivos, y cuyas horas de trabajo se pagan al doble que las horas de un trabajador fijo. Por último, se permiten pequeñas desviaciones de un turno de trabajo por cada tipo de turno ($T = 1$) entre los diferentes trabajadores fijos de un equipo.

3.1.2 Resolución

La descripción propuesta abstrae el problema como una entidad que recibe a nd, nt, ntw, er, ef, ws, abs, dc y a T como parámetros de entrada, computando como resultado la pareja $(timetabling, eh)$. $timetabling$ es una asignación de turnos representada mediante una matriz $w \times nd$, donde cada posición (i, j) representa el turno asignado en el día j al trabajador i (por ejemplo, en la instancia usada como ejemplo, $timetabling$ es una matriz 13×7). eh representa la cantidad total de horas extra de dicha asignación.

Una forma intuitiva de modelar el problema es usar a $timetabling$ como el conjunto de variables $\mathcal{FD}$, usando a dc y ws para determinar el dominio inicial de las variables en cada uno de esos días. Sin embargo, el hecho de que solo un equipo trabaje cada día, y de que los equipos roten (haciendo que cada equipo trabaje cada nt días) produce fuertes dependencias entre las variables de $timetable$: Tan pronto como se selecciona a un trabajador fijo w_k (perteneciente al equipo t_i) para trabajar el día d_j (asignando a este a un turno de trabajo $s_{jk} > 0$), entonces se puede concluir que t_i es el equipo seleccionado para trabajar en dicho día d_j, así como en los días d_{j+nt}, $d_{j+(2*nt)}$ y sucesivos. Esto excluye a los otros equipos de trabajo para estos días, al tiempo que excluye al equipo t_i de trabajar en los días d_{j+1}, $d_{j+(nt-1)}$, $d_{j+(nt+1)}$, $d_{j+(2*nt-1)}$ y sucesivos. La figura 8 presenta estas dependencias para la instancia usada como ejemplo, mostrando las implicaciones de asignar $timetabling_{1,2} = 20$. Por lo tanto, aunque estas dependencias podrían ser perfectamente modeladas (por ejemplo, mediante el uso de restricciones proposicionales de implicación) un enfoque más eficiente para modelar el problema es utilizar $Table$ en lugar de $timetabling$. $Table$ es una matriz $(ntw+1) \times nd$ donde, por cada d_j, las primeras ntw filas representan a los trabajadores fijos del equipo t_i seleccionado para trabajar en d_j, y la fila $(ntw + 1)$ representa a ew. Para utilizar esta aproximación a la resolución se necesitan varias traducciones bidireccionales entre las representaciones de $timetabling$ y $Table$. Estas requieren un registro adicional tda (de asignaciones de equipos a días), indicando qué equipo trabaja cada día. La figura 9 presenta esta traducción para la instancia usada como ejemplo, donde las cuatro primeras filas representan a {w_1, w_2, w_3, w_4} en los días 1, 4 y 7,

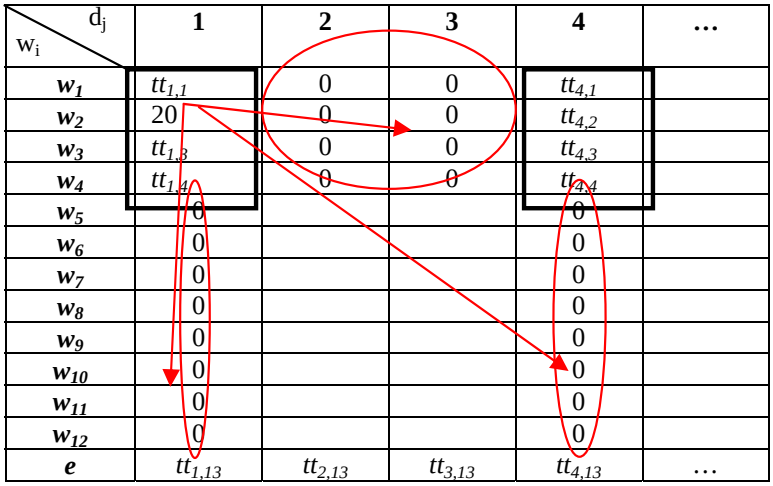

Figura 8. Dependencias de los equipos en $timetabling$

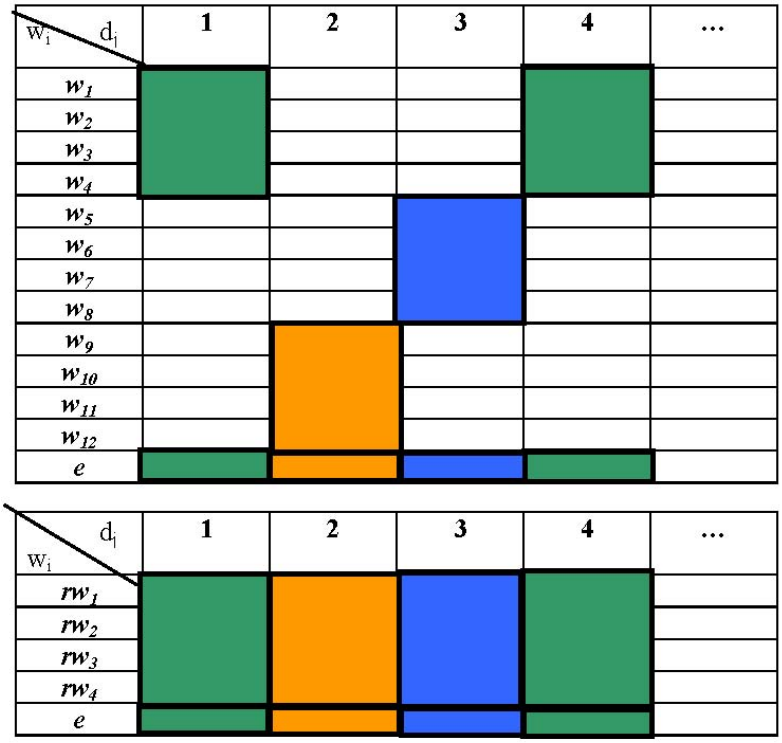

Figura 9. Traducción entre $timetabling$ y $Table$

a {w_9, w_{10}, w_{11}, w_{12}} en los días 2 y 5, y a {w_5, w_6, w_7, w_8} en los días 3 y 6.

Como se puede ver, una única $Table$ no permite explorar todo el espacio de búsqueda de $timetabling$, sino únicamente el subconjunto del espacio asociado al valor

concreto de tda seleccionado. Por lo tanto, la solución encontrada en $Table$ solo puede considerarse subóptima, en el sentido de que es solo es óptima con respecto a dicho subespacio de $timetabling$. De hecho, los nt equipos dan lugar a un total de $nt!$ posibles valores para $tda \equiv \{tda_1, \ldots, tda_{nt!}\}$ (cada uno de ellos con su $Table$ a explorar asociada). Para encontrar la solución óptima no se deben explorar a todas las posibles $Table$, sino únicamente a aquellas asociadas a valores de tda satisfactibles. Más específicamente, se dice que una tda_i concreta es satisfactible si, para cada día, hay suficientes trabajadores disponibles para llevar a cabo los turnos de trabajo requeridos (contando además con que nunca se seleccione a ew para trabajar más de uno de cada er días consecutivos). En este contexto, encontrar un tda_i insatisfactible permite ahorrar la exploración de un $1/nt!$ del espacio de búsqueda de $timetabling$. En la instancia usada como ejemplo, solo dos de los seis tda son satisfactibles ya que, debido a las ausencias proporcionadas por abs, t_1 es el único equipo que puede trabajar en d_1. Las dos asignaciones satisfactibles son, por tanto, asignar t_2 (respectivamente t_3) a d_2 y t_3 (respectivamente t_2) a d_3.

Además, los nt equipos están vinculados puesto que, en caso necesario, solo se puede seleccionar a ew para trabajar uno de cada er días consecutivos. Sin embargo, como un tda satisfactible implica cumplir esta restricción de descanso para ew, su $Table$ asociada se puede dividir en nt subproblemas independientes (y exponencialmente más sencillos de resolver). En la instancia usada como ejemplo, la $Table$ asociada a la figura 9 se divide en tt_1 (columnas 1, 4 y 7), tt_2 (columnas 2 y 5) y tt_3 (columnas 3 y 6).

En resumen, el modelo más intuitivo basado en $timetabling$ requiere $w \times nd$ variables (en la instancia usada como ejemplo $13 \times 7 \equiv 91$). Suponiendo que, debido a los turnos de trabajo requeridos, el dominio inicial de cada variable fuese {0, 20, 22, 24}, entonces el espacio de búsqueda inicial contendría $4^91 \equiv 6,12 * 10^{54}$ candidatos. Por lo tanto, el enfoque de resolución basado en tda mejora la eficiencia de resolución mediante:

- El uso de $Table$, cuyas $(ntw+1) \times nd$ variables permiten ahorrar $ntw*(nt-1) \times nd$ variables con respecto a la aproximación basada en $timetabling$. En la instancia usada como ejemplo, $Table$ es de 5×7, ahorrando un 61% de las variables de $timetabling$, y reduciendo así el espacio de búsqueda a $4^{0,39*91} \equiv 2,32 * 10^{21}$.
- La división de $Table$ en nt subproblemas independientes, y exponencialmente más sencillos de resolver. En la instancia usada como ejemplo, hay tres equipos independientes (en la figura 9 se pueden identificar mediante las columnas verdes, naranjas y azules, respectivamente). Por lo tanto, el espacio de búsqueda de cada uno de estos subproblemas es de $4^{0,13*91} \equiv 1,32 * 10^7$. Como los tres subproblemas deben ser resueltos, el espacio de búsqueda explorado resulta ser $3,96 * 10^7$.
- La exploración de únicamente aquellas $Table_i$ asociadas a tda_i satisfactibles. En la instancia usada como ejemplo, solo dos tda de las $3! = 6$ posibles son satis-

factibles. Por lo tanto, para encontrar la solución óptima el espacio de búsqueda explorado resulta ser $7,92 * 10^7$, que es mucho más pequeño que el espacio original de $6,12 * 10^{54}$ candidatos.

3.2 BPP

La segunda aplicación de $\mathcal{TOY}(\mathcal{FD})$ consiste en un análisis empírico acerca de la complejidad computacional del BPP clásico. Dicho análisis resuelve un conjunto de instancias generadas paramétricamente para los que aplica tanto a las heurísticas MAXREST, FIRSTFIT, BESTFIT y NEXTFIT, como a dos modelos CP($\mathcal{FD}$) equivalentes de Gecode y $\mathcal{TOY}(\mathcal{FD}g)$. Este conjunto de instancias BPP se basa en el modelo estadístico paramétrico de Weibull, cuya función de densidad de probabilidad es particularmente interesante, ya que permite modelar con precisión diferentes aspectos de instancias BPP de la vida real. Las conclusiones obtenidas en el análisis proporcionan una base para el futuro desarrollo de resolutores *ad hoc* que permitan resolver intancias BPP generalizadas. Estos resolutores proporcionarán técnicas híbridas entre las heurísticas y la resolución de restricciones CP($\mathcal{FD}$). Así, se basarán en la estructura de la instancia (los parámetros concretos que la generan) para determinar una configuración más eficiente (respecto a la técnica más adecuada a aplicar para resolver dicha instancia).

La subsección 3.2.1 presenta el modelo de Weibull, así como su aplicabilidad para ajustar instancias BPP reales. Las subsecciones 3.2.2 y 3.2.3 presentan la generación del conjunto de instancias y de los archivos de procesamiento por lotes (para la resolución de dicho conjunto mediante las diferentes técnicas), respectivamente.

3.2.1 Modelo paramétrico de Weibull

La conocida distribución contínua de Weibull [201] se define por dos parámetros: la forma ($k > 0$) y la escala ($\lambda > 0$). La figura 10 presenta la función de densidad de probabilidad, $f(x; \lambda, k)$, de una variable aleatoria x distribuida acorde a Weibull. Los parámetros dan lugar a una gran flexibilidad, permitiendo representar a distribuciones de diferentes dominios de aplicación, entre los que se incluye BPP. Las figuras 11 y 12 presentan varios ejemplos de las diferentes distribuciones que se pueden obtener con Weibull. En la figura 11 se muestran cuatro distribuciones, donde el parámetro k toma valores pequeños ($0,5$, $1,0$, $1,5$ y $5,0$). Como se puede ver, las distribuciones obtenidas

$$f(x; \lambda, k) = \begin{cases} \frac{k}{\lambda} \cdot (\frac{x}{\lambda})^{k-1} \cdot e^{-(x/\lambda)^k} & x \geq 0, \\ 0, & \text{otherwise} \end{cases}$$

Figura 10. Función de densidad de probabilidad de Weibull

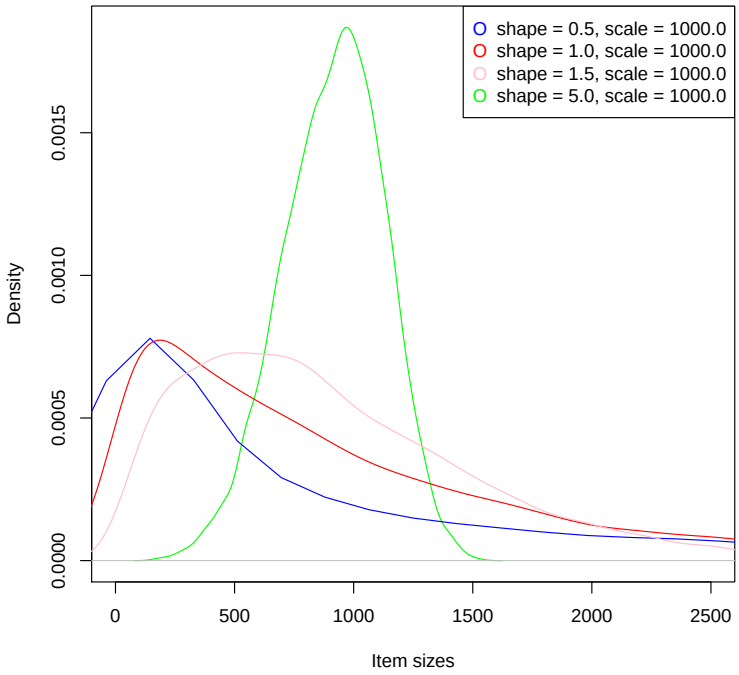

Figura 11. Distribuciones de Weibull para valores pequeños de k

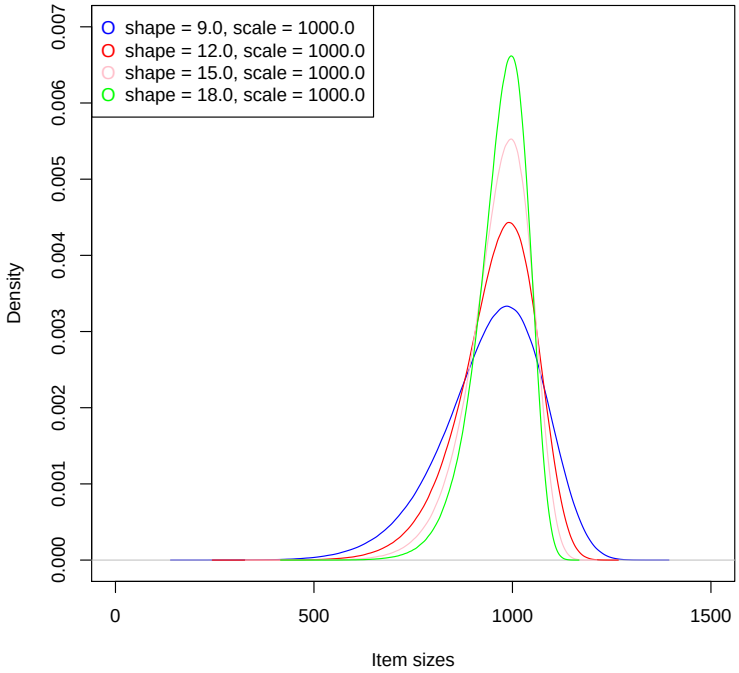

Figura 12. Distribuciones de Weibull para valores grandes de k

son muy diferentes, algunas proporcionando valores que se expanden varios órdenes de magnitud con respecto al parámetro λ (aunque con una probabilidad muy baja). La figura 12 muestra valores más grandes de k ($9,0$, $12,0$, $15,0$ y $18,0$), lo que permite observar que las distribuciones obtenidas con Weibull presentan una menor variación a medida que k aumenta.

Se ha comprobado que un conjunto de instancias BPP, basadas en la distribución de Weibull, representa con éxito a instancias BPP generalizadas provenientes de la industria de la optimización en centros de datos [45]. Para ello, se ha utilizado a la plataforma de computación estadística R [8], comprobando el ajuste que proporciona la distribución de Weibull a los datos de dichas instancias reales, mediante el uso de máxima verosimilitud (del inglés *Maximum Likelihood Fitting: MLF*). Como ejemplo, la figura 13 muestra el ajuste de Weibull (círculos en verde) para los datos de una de estas instancias (línea en negro). Este análisis visual puede complementarse mediante gráficos Q-Q (del inglés *Quantile-Quantile plots*). Estos gráficos proporcionan una herramienta simple para determinar si dos conjuntos de datos provienen de la misma distribución subyacente. En una gráfico Q-Q, cada punto corresponde a un determinado segmento de ambos conjuntos de datos. Si, en la gráfica resultante el conjunto de puntos se encuandran sobre la diagonal ascendente (línea de pendiente 1, o de 45 grados) entonces se puede concluir que ambos conjuntos de datos provienen de la misma distribución subyacente. La figura 14 presenta un ejemplo del gráfico Q-Q obtenido para la misma instancia real utilizada anteriormente.

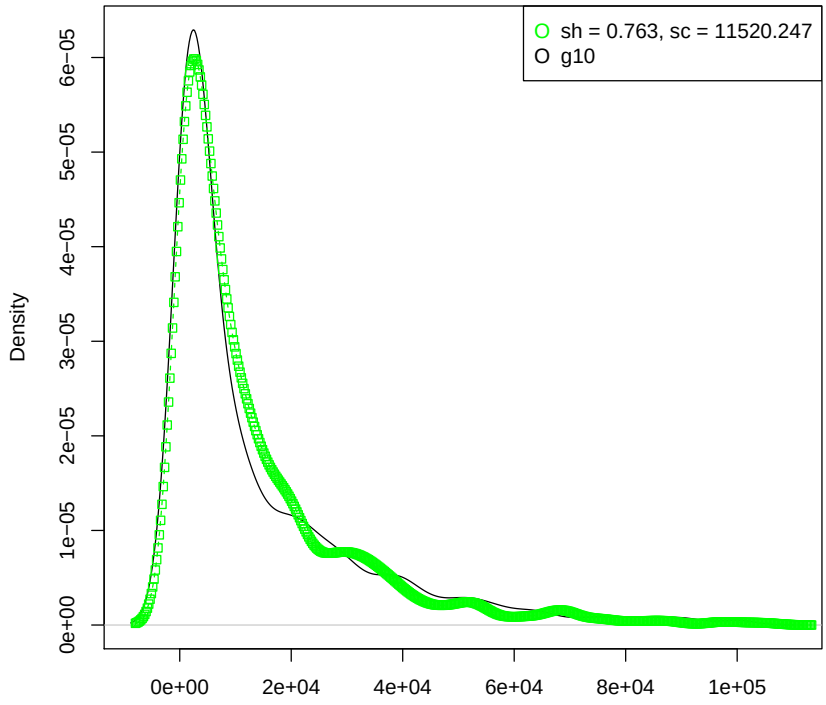

Figura 13. Ajuste Weibull por MLF

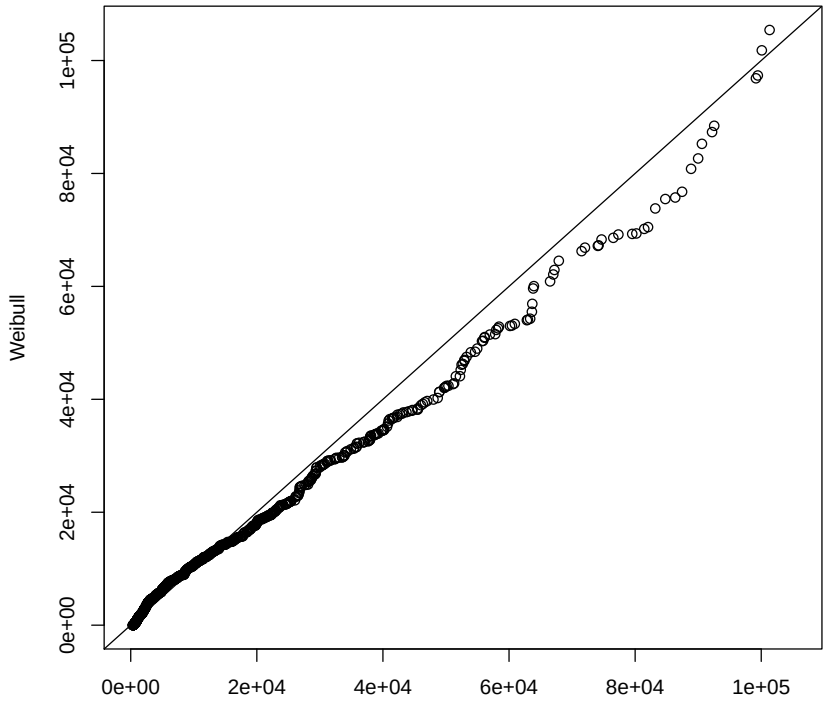

Figura 14. Ajuste Weibull por gráficos Q-Q

Se puede comprobar que la distribución de Weibull computada por MLF se ajusta a los datos de la instancia real. En particular, para los valores que van desde 0 a 20.000 (que representan la mayoría de la masa de probabilidad) el ajuste es muy preciso. Por otro lado, para los valores atípicos, que abarcan varios órdenes de magnitud, este ajuste es mucho peor. Sin embargo, esto tiene sentido, puesto que la probabilidad de cada uno de estos valores es prácticamente 0.

3.2.2 Generación del conjunto de instancias

El conjunto de instancias BPP se genera mediante la biblioteca Boost [30]. Esta es una API C++ que incluye definiciones de tipo para una distribución de Weibull, parametrizada por un generador de números aleatorios, la forma (k) y la escala (λ). También se proporciona la opción de iterar por los valores generados por la distribución. Para generar el conjunto de instancias, se fija λ al valor 1000, de manera que los tamaños de los elementos se expanden tres órdenes de magnitud. Para k se considera un amplio rango de valores $[0,1,\ 0,2,\ \ldots,\ 19,9]$, dando lugar a 199 combinaciones diferentes de los parámetros (k, λ), de ahora en adelante categorías. Volviendo sobre las figuras 11 y 12, se observa que estas categorías dan lugar a distribuciones muy diferentes.

Para evitar la aleatoriedad en los experimentos se generan 100 instancias para cada categoría, lo que da lugar a la generación de 19.900 instancias. Cada una de ellas contiene 100 elementos, cuyos tamaños se almacenan en un archivo asociado (en el que

se encuentran representados en un orden decreciente). Este archivo puede ser identificado unívocamente mediante los parámetros (k, λ) y el índice ($0, \ldots, 99$) dentro de la categoría. Además, los archivos se enumeran de 1 a 19.900. Así, por ejemplo, las 100 instancias generadas para $k = 2,5$ están representadas en $inst\_2401(2.5, 1000, 0).data$, ..., $inst\_2500(2.5, 1000, 99).data$. A modo de ejemplo, la instancia $inst\_2405(2.5, 1000, 4).data$ contiene los siguientes tamaños de elementos: $\{2001, 1699, 1657, 1647, 1591, 1556, 1534, 1498, 1480, 1466, 1451, 1374, 1365, 1352, 1352, 1350, 1335, 1306, 1298, 1259, 1243, 1224, 1223, 1223, 1212, 1208, 1207, 1202, 1183, 1180, 1175, 1161, 1139, 1133, 1115, 1101, 1093, 1091, 1062, 1062, 1059, 1058, 1005, 981, 979, 970, 969, 955, 946, 941, 928, 923, 916, 911, 888, 854, 849, 844, 809, 808, 808, 803, 769, 753, 728, 716, 672, 670, 665, 656, 651, 622, 591, 588, 570, 567, 558, 554, 552, 538, 527, 527, 507, 503, 490, 450, 437, 402, 386, 371, 365, 355, 325, 321, 312, 297, 205, 193, 177, 135\}$.

Una vez fijado el conjunto de instancias, se han analizado 11 capacidades C diferentes para el contenedor, oscilando su tamaño entre 1,0 y 2,0 (con incrementos de 0,1) veces el tamaño del elemento más grande de la instancia a resolver. Por ejemplo, para resolver la instancia $inst\_2405(2.5, 1000, 4).data$, en los 11 experimentos considerados C ha tomado los valores 2001, 2202, 2402, 2602, 2802, 3002, 3202, 3402, 3602, 3802 y 4002, respectivamente.

3.2.3 Ejecución del conjunto de instancias

En primer lugar se presenta la configuración de los modelos C++ de Gecode y de las heurísticas. Despues se describen las pequeñas diferencias necesarias para configurar $\mathcal{TOY}(\mathcal{FD})$.

En C++, los modelos se compilan para generar ejecutables. Estos archivos ejecutables se pueden lanzar mediante un comando en una sesión de comandos del sistema operativo. Los modelos de Gecode y de las heurísticas son adaptados, de manera que estos reciben el nombre de la instancia de la que extraer los tamaños de los elementos (por ejemplo, $inst\_2405(2.5, 1000, 4).data$), así como el factor por el que multiplicar al tamaño del elemento más grande, para establecer el tamaño del contenedor (por ejemplo, $1,0$). Además, reciben el nombre del archivo donde almacenar la solución obtenida (por ejemplo, $inst\_2405(2.5, 1000, 4).data$).

El comando `bpp.exe 2405 "$models_path$/bpp_instances/inst_2405(2.5, 1000, 4).data" "$models_path$/Gecode/C++/bpp_solutions/inst_2405(2.5, 1000, 4).sol" 2.5 1.0 10000` resuelve la instancia $inst\_2405(2.5, 1000, 4).data$ usando el modelo Gecode. El archivo $.sol$ generado contiene una sola línea `ID = 2405 Solved = 1 Time = 218 Bins = 47`, indicando que la instancia 2405 se ha resuelto en 218 milisegundos mediante el uso de una cantidad óptima de 47 contenedores.

El comando `bpp.exe 2405 "$models_path$/bpp_instances/inst_2405(2.5, 1000, 4).data" "$models_path$/Heuristics/bpp_solutions/inst_2405(2.5, 1000, 4).sol" 1.0` resuelve la instancia $inst\_2405(2.5, 1000, 4).data$ utilizando

las heurísticas. El archivo $.sol$ generado contiene cuatro líneas, una por heurística, indicando la instancia que ha sido resuelta, el tiempo empleado y el número de contenedores usados.

```
ID = 2405 MaxRest: 0.000 48
ID = 2405 FirstFit: 0.000 47
ID = 2405 NextFit: 0.000 63
ID = 2405 BestFit: 0.000 47
```

Se genera un archivo de procesamiento por lotes (`.bat`), que contiene los 19.900 comandos para ejecutar cada una de las instancias del conjunto. La ejecución de un archivo de procesamiento por lotes se llamará también *sesión*. Como para cada instancia se consideran 11 configuraciones de C diferentes (variando entre 1,0 y 2,0, con incrementos de 0,1), se utilizan 11 archivos de procesamiento por lotes. Por ejemplo, el archivo `bpp_session_1.0.bat` da lugar a una sesión para ejecutar el conjunto de instancias, considerando para cada una de ellas que los contenedores tienen un tamaño igual al elemento más grande de la instancia.

La ejecución de una sesión en $\mathcal{TOY}(\mathcal{FD}g)$ requiere unos pocos cambios con respecto al proceso descrito para Gecode y las heurísticas. En primer lugar, $\mathcal{TOY}(\mathcal{FD})$ no se compila a un lenguaje máquina, por lo que cada comando que ejecuta una instancia debe ser considerado como un objetivo $\mathcal{TOY}(\mathcal{FD})$ (que se ejecuta dentro de una sesión del sistema). En segundo lugar, un objetivo $\mathcal{TOY}(\mathcal{FD})$ no puede leer de un archivo, por lo que este debe incluir a la lista de elementos y al tamaño del contenedor como argumentos. En tercer lugar, se modifica el flujo de salida del sistema, de manera que se escriba la solución obtenida en un archivo unívocamente identificado (por el id de la instancia que se ha ejecutado). En este contexto, el objetivo `bpp 2405 2001 [2001, 1699, 1657, 1647, 1591, 1556, 1534, 1498, 1480, 1466, 1451, 1374, 1365, 1352, 1352, 1350, 1335, 1306, 1298, 1259, 1243, 1224, 1223, 1223, 1212, 1208, 1207, 1202, 1183, 1180, 1175, 1161, 1139, 1133, 1115, 1101, 1093, 1091, 1062, 1062, 1059, 1058, 1005, 981, 979, 970, 969, 955, 946, 941, 928, 923, 916, 911, 888, 854, 849, 844, 809, 808, 808, 803, 769, 753, 728, 716, 672, 670, 665, 656, 651, 622, 591, 588, 570, 567, 558, 554, 552, 538, 527, 527, 507, 503, 490, 450, 437, 402, 386, 371, 365, 355, 325, 321, 312, 297, 205, 193, 177, 135] 10000 true == Result` resuelve la instancia $inst\_2405(2.5, 1000, 4).data$ usando $\mathcal{TOY}(\mathcal{FD}g)$. Este objetivo vincula $Result -> 47$, y genera el archivo $inst\_2405.sol$, que contiene como única línea `ID = 2405 Solved = 1 Time = 202 Bins = 47` (equivalente a la computada por Gecode).

Para ejecutar el conjunto de instancias, $\mathcal{TOY}(\mathcal{FD})$ no puede hacer uso de archivos de procesamiento por lotes, por lo que utiliza una función indeterminista para simular el comportamiento de estos. Es decir, un archivo de sesión $\mathcal{TOY}(\mathcal{FD}g)$ (por ejemplo, `bpp_session_10.toy`, que utiliza $C = 1.0$) contiene solo una función indetermi-

nista bpp_session:: bool. Esta función no recibe ningún argumento, y contiene 19.900 reglas condicionales. Cada una de estas reglas devuelven true si y solo si la ejecución de una instancia concreta de conjunto tiene éxito (por ejemplo, la regla 2405 de bpp_session devuelve true si el objetivo de ejecutar la instancia 2405 tiene éxito). Todo el conjunto de instancias se ejecuta mediante la inclusión de la función bpp_session en el modelo $\mathcal{TOY}(\mathcal{FD}g)$ y la ejecución del objetivo bpp_session == true (solicitando además al sistema que compute todas las soluciones). Más específicamente, $\mathcal{TOY}(\mathcal{FD})$ sufre problemas de memoria cuando la función contiene más de 300 reglas, por lo que para ejecutar el conjunto completo de instancias bpp_session se divide en bloques de 250 reglas, y cada bloque se ejecuta en una sesión $\mathcal{TOY}(\mathcal{FD}g)$.

4 $\mathcal{TOY}(\mathcal{FD})$ comparado con otros sistemas CP($\mathcal{FD}$)

Aunque CFLP($\mathcal{FD}$) es un paradigma adecuado para abordar a un CSP o COP, la literatura carece de tantas aplicaciones reales como las existentes para otros paradigmas CP($\mathcal{FD}$) consolidados, como CP($\mathcal{FD}$) algebraicos, C++ CP($\mathcal{FD}$) o CLP($\mathcal{FD}$). En esta tercera parte de la investigación se utilizan dos COP, el CP($\mathcal{FD}$) clásico de Golomb y el industrial de ETP, para realizar una comparativa en profundidad acerca del modelado y resolución de estos COP en diferentes sistemas CP($\mathcal{FD}$) de vanguardia. En concreto se consideran los sistemas CP($\mathcal{FD}$) algebriacos Minizinc e ILOG OPL, los sistemas C++ CP($\mathcal{FD}$) Gecode e ILOG Solver, los sistemas CLP($\mathcal{FD}$) SICStus Prolog y SWI-Prolog, y los sistemas CFLP($\mathcal{FD}$) PAKCS y $\mathcal{TOY}(\mathcal{FD})$ (este último con sus tres versiones $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ y $\mathcal{TOY}(\mathcal{FD}s)$). Las secciones 4.1 y 4.2 presentan los principales resultados obtenidos en la comparativa de modelado y resolución, respectivamente.

4.1 Comparativa de modelado

El resolutor de restricciones es transparente para el usuario en los sistemas CP($\mathcal{FD}$) algebraicos, C++ CP($\mathcal{FD}$) y CLP($\mathcal{FD}$), por lo que también lo es su gestión. En el caso de los sistemas C++ CP($\mathcal{FD}$), estos desarrollan modelos específicos para un determinado resolutor, por lo que se hace explícita la gestión de las variables de decisión, las restricciones, la función objetivo, el almacén de restricciones, la propagación de restricciones, el motor de búsqueda y el control que este hace de la exploración. De igual modo, es necesario gestionar la recolección de basura de todos estos elementos. Más específicamente, en Gecode la noción de resolutor de restricciones está representada mediante un objeto `Space`, el cual se programa por herencia, utilizando subclases dedicadas, cuyo constructor de clase contiene la formulación del problema a resolver. En ILOG Solver la noción de resolutor de restricciones está representada mediante un objeto `IloSolver`, pero el problema se formula en la capa genérica de modelado de ILOG Concert, que incluye un objeto para representar explícitamente al almacén de restricciones. Obviamente, esta capa ILOG Concert también incluye un proceso de traducción entre las variables y restricciones genéricas impuestas en el almacén y las variables y restricciones especializadas con las que trabajará `IloSolver`. Finalmente, para mostrar la solución obtenida, los sistemas C++ CP($\mathcal{FD}$) requieren métodos específicos de `Space` e `IloSolver`, que permitan acceder a los valores computados para las variables. En los sistemas CP($\mathcal{FD}$) algebraicos, CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$), como las variables son declaradas libremente, estas se pueden utilizar directamente para mostrar sus valores obtenidos.

Cuando la formulación de un problema involucra múltiples etapas, los sistemas CP($\mathcal{FD}$) algebraicos requieren varios modelos (almacenados en archivos independientes). Este es el caso de ETP, para el que los sistemas CP($\mathcal{FD}$) algebraicos utilizan cuatro

archivos, ya que tanto la etapa $team\_assign$ como tt_solve requieren un archivo independiente. En el primero de los casos, esto es debido a que existen variables que se deben tratar como variables de decisión en $team\_assign$, pero como parámetros en tt_split y tt_map. En el caso de tt_solve, la etapa se debe aislar para poder explotar la independencia de cada equipo. Un archivo de procesamiento externo coordina la ejecución de los diferentes modelos, generando también los argumentos de entrada para cada uno de ellos. La programación de este archivo de procesamiento representa una dificultad añadida (ya que es totalmente independiente del modelado de un problema CP($\mathcal{FD}$)). Para el resto de paradigmas, el modelo de ETP está contenido en un solo archivo. En los sistemas C++ CP($\mathcal{FD}$), las etapas tt_split y tt_map se implementan fácilmente mediante el uso de las abstracciones de C++, pero las etapas $team\_assign$ y la etapa tt_solve de cada equipo requieren un resolutor de restricciones dedicado. Esto es obligatorio ya que los resolutores C++ CP($\mathcal{FD}$) no soportan el almacenamiento de nuevas restricciones (como las impuestas en tt_solve) cuando estos están en medio de una exploración de búsqueda (como la que se inicia en $team\_assign$). Mientras que en Gecode la abstracción de un resolutor de restricciones dedicado para $team\_assign$ y tt_solve da lugar a diferentes subclases de `Space`, en ILOG Solver todos los resolutores se implementan como objetos `IloSolver`. En los sistemas CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$), el uso de un razonamiento por modelos permite abordar fácilmente la formulación propuesta en el algoritmo p_tt. Las diferentes etapas son coordinadas mediante una simple emumeración en orden de estas, con una primitiva $labeling$ colocada al final de $team\_assign$ y tt_solve, para asegurar una correcta implementación de la arquitectura de p_tt. Obviamente, tanto los sistemas CLP($\mathcal{FD}$) como CFLP($\mathcal{FD}$) necesitan gestionar internamente varios resolutores de restricciones (para $team\_assign$ y tt_solve en cada equipo). Sin embargo, como estos sistemas abstraen el concepto de resolutor de restricciones, su gestión es transparente al usuario.

Mientras que los sistemas C++ CP($\mathcal{FD}$), CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$) utilizan estructuras de datos dinámicas (como vectores y listas), los sistemas CP($\mathcal{FD}$) algebraicos se basan en *arrays* estáticos. Por ello, estos últimos necesitan parámetros de entrada adicionales que les permitan determinar por adelantado la cantidad de elementos que deberán contener. En tt_split se utilizan tanto arrays bidimensionales como tridimensionales que permiten configurar los parámetros de entrada y la estructura de los diferentes equipos que serán posteriormente resueltos en tt_solve. Como estos arrays tienen una longitud fija, se necesitan variables y restricciones $\mathcal{FD}$ adicionales para representar días extra en los equipos con un menor número de días. Por otra parte, estos arrays proporcionan tanto acceso como indexación directo para cada uno de sus elementos, mientras que en los sistemas CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$) se necesitan variables adicionales (e incluso predicados y funciones dedicados) para acceder e indexar los elementos de las listas.

El ahorro de variables $\mathcal{FD}$ permite reducir la red de restricciones impuesta sobre el almacén. Tomando como ejemplo las variables equivalentes de tt y $trans\_tt$

en ETP, se ha comprobado que los sistemas CP($\mathcal{FD}$) algebraicos no permiten reducir el número de variables impuestas, ya que tt y $trans\_tt$ se encuentran declaradas en dos arrays bidimensionales independientes (con sus variables equivalentes posteriormente relacionadas mediante restricciones explícitas de igualdad). En los sistemas C++ CP($\mathcal{FD}$), CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$), de cara al modelado, estas variables equivalentes dan lugar a dos representaciones diferentes. Sin embargo, de cara a la resolución, ambas se encuentran representadas mediante una única variable común impuesta sobre el almacén. Para ello, en los sistemas C++ CP($\mathcal{FD}$) es suficiente con inicializar cada $trans\_tt_{ji}$ usando a tt_{ij} como su parámetro. En concreto en Gecode, tt y $trans\_tt$ forman parte de los atributos de la clase `StageIII`. Sin embargo, estos se representan como vectores unidimensionales (a diferencia de los vectores bidimensionales usados en ILOG Solver y que resultan una opción mucho más intuitiva). Esta aproximación unidimensional implica una dificultad añadida al modelado puesto que exige recomputar todos los índices de las variables que intervienen en las restricciones impuestas. Sin embargo, esto permite clonar tanto tt como $trans\_tt$ por parte del constructor de copia mediante una única operación atómica. El uso de vectores bidimensionales habría exigido n operaciones (una por cada día de trabajo del equipo), penalizando el rendimiento del propio método de copia y el del propio tiempo de resolución del problema. Como puede apreciarse, este uso de vectores unidimensionales representa un matiz de bajo nivel, y demuestra que la falta de abstracción en C++ CP($\mathcal{FD}$) puede dar lugar a una ruptura del denominado aislamiento entre modelado y resolución de un problema. En los sistemas CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$), tt se construye mediante la generación de nuevas variables lógicas, y $trans\_tt$ se construye a partir de tt mediante unificación de patrones. Mientras que el resolutor $\mathcal{H}$ es el encargado de la unificación de dos variables lógicas, la igualdad de dos variables $\mathcal{FD}$ da lugar a la imposición de una restricción $\mathcal{FD}$ sobre el almacén. Por lo tanto, la unificación entre las variables equivalentes de tt y $trans\_tt$ debe realizarse antes de imponer cualquier restricción $\mathcal{FD}$ sobre estas. Esto contradice la naturaleza puramente declarativa de los sistemas CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$), ya que el orden en que se formule el problema se convierte en relevante para la eficiencia de la resolución.

El uso de expresiones de *agregación* permite a los sistemas CP($\mathcal{FD}$) algebraicos declarar bloques de restricciones en una sola línea de código, de manera mucho más elegante que en el caso de los bucles imperativos de C++ CP($\mathcal{FD}$) o de los procesos recursivos de CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$). Sin embargo, como ni MiniZinc ni ILOG Solver incluyen la noción de contador para estos bloques, la declaración de algunas restricciones resulta mucho más compleja (especialmente cuando estas implican una relación compleja entre los índices de las variables involucradas). En los sistemas CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$), el uso de variables auxiliares declaradas al vuelo facilita la especificación de algunas de estas restricciones. Además, en CFLP($\mathcal{FD}$), el uso del orden superior y la notación funcional permite evitar la declaración explícita de estas variables auxiliares.

En cuanto a la propagación de restricciones, los sistemas CP($\mathcal{FD}$) algebraicos solo

soportan el modo por lotes, y el sistema CFLP($\mathcal{FD}$) PAKCS solo soporta el modo incremental. En el caso de C++ CP($\mathcal{FD}$), por defecto, el modo de propagación es por lotes, pero se puede modificar a los modelos para que estos soporten el modo incremental, mediante el uso de variables extra que controlan la satisfiabilidad del almacén de restricciones tras cada nueva restricción impuesta. En el caso de CLP($\mathcal{FD}$) el modo de propagación es incremental de forma predeterminada, pero se puede modificar a los modelos para que estos soporten el modo por lotes mediante el uso de variables extra que congelen el almacenamiento de las restricciones impuestas. Finalmente, $\mathcal{TOY}(\mathcal{FD})$ contiene primitivas explícitas para establecer un modo u otro de propagación, permitiendo aplicar de forma trivial diferentes modos de propagación a diferentes partes del modelo.

En cuanto a la exploración de búsqueda, además del control sobre esta, la declaración debe especificar el conjunto de variables, el orden en que estas se etiquetan, el orden en que se prueban los diferentes valores del dominio de cada variable y, finalmente, la función de coste. Los sistemas CP($\mathcal{FD}$) algebraicos, CLP($\mathcal{FD}$) y CFLP($\mathcal{FD}$) proporcionan primitivas expresivas que permiten especificar todos estos aspectos en una línea de código. En Gecode, la especificación es de más bajo nivel, pues está asociada a la exploración del árbol que se va a realizar. Los nodos del árbol se abstraen mediante objetos `Space`. Un motor de búsqueda se asocia a un cierto `Space`, para controlar la exploración de búsqueda. El constructor de copia del `Space` debe programarse explícitamente (puesto que el motor lo utiliza para clonar `Spaces`, paso imprescindible para la realización de la recomputación híbrida). También debe programarse explícitamente la función de coste, ya que esta se encarga de añadir dinámicamente nuevas restricciones que permiten continuar la búsqueda para encontrar nuevas soluciones *mejores* que la actual. De este modo, forzando al motor a buscar todas las soluciones, se puede garantizar que la última solución encontrada será la óptima. ILOG Solver requiere varias primitivas para especificar la búsqueda que se componen para generar el *demonio* final que será ejecutado por `IloSolver`. Sin embargo, no se requiere un control de la búsqueda ni la programación de métodos específicos.

El modelado de los COP propuestos explota en gran medida la expresividad que ofrece cada paradigma.

En el caso de los sistemas CP($\mathcal{FD}$) algebraicos, estas características incluyen:

- El uso de conjuntos (`set`) como una estructura de datos básica, lo que incluye también el uso de arrays de conjuntos.
- El uso de rangos de enteros `a..b`, que indican el conjunto `{a,(a+1),(a+2),..,b}`. La posibilidad de seleccionar a todos los elementos de un rango, o tan solo a aquellos que cumplen una determinada condición.
- El uso de expresiones `sum`, para contabilizar la cantidad de elementos de una estructura de datos que cumplen una condición.

- La carencia, en MiniZinc de un método para obtener la cardinalidad de un conjunto. Sin embargo, MiniZinc sí que permite acceder al *i*-ésimo elemento del conjunto a, simplemente usando `a[i]`. Además, permite comprobar si un elemento e pertenece al conjunto a, simplemente usando `(e in a)`, y permite transformar este resultado en un entero, utilizando la primitiva `bool2int`.
- La necesidad de convertir a cualquier parámetro v que vaya a mostrarse como parte de la solución en el bloque `output`, en una variable de decisión ($\mathcal{FD}$). Esto también implica, como efecto colateral, la conversión de cualquier otro parámetro v' asociado a v.
- El uso de restricciones de implicación c_1 -> c_2. En ellas, c_1 debe soportar reificación y c_2 se impone únicamente cuando c_1 se reduce a `true`.
- El uso de variables locales.

En el caso de los sistemas C++ CP($\mathcal{FD}$), estas características incluyen:

- El uso de un estilo de programación imperativo, que controla explícitamente el orden de ejecución del modelo.
- El uso de métodos, que proporcionar una mayor modularidad en el código.
- El paso de parámetros por valor y por referencia, permitiendo un uso más eficiente de la memoria.
- El uso de la biblioteca estándar, que abstrae la gestión de `vector`.

En el caso de los sistemas CFLP($\mathcal{FD}$), estas características incluyen:

- El uso de funciones de orden superior, que dan lugar a una formulación más concisa.
- El uso de variables extras en el lado derecho de una regla de función.
- El uso de la evaluación perezosa, que permite evaluar los argumentos de una función bajo demanda (el uso de la llamada por valor, utilizado en LP, en contraste con el uso de la llamada bajo demanda, utilizado en FP [147]).
- El uso de declaraciones de tipos, que permite el desarrollo de programas más seguros y mantenibles. Es importante señalar que la declaración de tipo es opcional, ya que el sistema siempre infiere un tipo para cada función, tanto si este se declara como si no. Finalmente, el sistema también soporta argumentos polimórficos.
- El uso de la aplicación parcial.
- El uso de encaje de patrones, que permite discriminar la regla de una función a aplicar.

En el caso de los sistemas CLP($\mathcal{FD}$), algunas de sus características son directamente comparadas con las de los sistemas CFLP($\mathcal{FD}$):

- La formulación de los predicados resulta ser menos compacta que la de funciones, debido a la ausencia de orden superior.
- La cantidad de variables utilizada. En concreto la ausencia de funciones de orden superior hace explícita la declaración de variables auxiliares, que en CFLP($\mathcal{FD}$) se pueden declarar de manera implícita. Por otro lado, en CFLP($\mathcal{FD}$), gracias a los argumentos polimórficos, una misma función de orden superior se puede aplicar en distintos escenarios, mientras que en CLP($\mathcal{FD}$) se requiere la creación de un predicado específico para cada uno de estos escenarios.
- El uso de argumentos de entrada/salida.
- La posible aplicación de encaje de patrones también en el resultado de un cómputo.
- El uso del operador de corte (!), que impide la evaluación futura de las claúsulas restantes del predicado (lo cual, en caso de hacer encaje de patrones, dispararía su ejecución). Dicho operador no está soportado en CFLP($\mathcal{FD}$), por lo que, para evitar que múltiples reglas puedan hacer encaje de patrones, todas ellas deben ser mutuamente excluyentes. En cualquier caso, la ausencia de un operador de corte no se debe ver como un inconveniente, ya que este representa un mecanismo no declarativo (pues convierte en relevante el orden en que se definen las cláusulas de un predicado).

4.2 Comparativa de resolución

Se ha establecido un marco común para la realización de los experimentos teniendo en cuenta el computador y las versiones de cada sistema utilizados, así como el algoritmo seguido para resolver los problemas, el conjunto de variables y restricciones usado (incluyendo a las restricciones globales), el modo de propagación, la estrategia de búsqueda y la medición del tiempo empleado para obtener la solución. Se ha justificado cualquier ligera desviación de cada sistema con respecto a esta configuración común.

Se han considerado a las instancias G-9 y ETP-7 (resueltas en décimas de segundo), G-10 y ETP-15 (resueltas en segundos) y, finalmente, G-11 y ETP-21 (resueltas en minutos). Como era de esperar, a medida que las instancias aumentan lo suficiente, la exploración de búsqueda se convierte en el factor clave en el tiempo empleado para solucionar el problema. En este contexto, G-9, G-10, G-11 y ETP-21 se pueden clasificar como instancias de *solo búsqueda* (del inglés *just search: js*), ya que, para ellas, las tres versiones de $\mathcal{TOY}(\mathcal{FD})$ dedican más del 98% de su tiempo de resolución a la búsqueda, convirtiendo a dicha resolución en puramente CP($\mathcal{FD}$) dependiente. Por otra

parte, ETP-7 y ETP-15 se pueden clasificar como instancias de *diferentes factores* (del inglés, *different factors: df*), ya que, para ellas, las tres versiones de $\mathcal{TOY}(\mathcal{FD})$ dedican entre un 0%-98% de su tiempo de resolución a la búsqueda, lo que convierte a dicha resolución en dependiente tanto de CP($\mathcal{FD}$) como de la sobrecarga inherente a tareas de cada paradigma.

Los resultados de resolución generales ponen de manifiesto que para las instancias *js* hay un orden claro en el rendimiento alcanzado por los diferentes resolutores de restricciones, pues los tres sistemas que utilizan un resolutor Gecode, ILOG Solver y SICStus `clpfd` alcanzan respectivamente las posiciones 1-3, 4-6, 7-9 (con SWI-Prolog en el décimo lugar). Para las instancias *df* este *ranking* entre los resolutores se rompe parcialmente: por un lado, SICStus e ILOG Solver obtienen un mejor posicionamiento para ETP-7 y ETP-15, así como PAKCS, $\mathcal{TOY}(\mathcal{FD}s)$ y SWI-Prolog, aunque estos últimos lo hacen solo para ETP-7. Por otro lado, MiniZinc y $\mathcal{TOY}(\mathcal{FD}i)$ obtienen un peor posicionamiento para ETP-7 y ETP-15, así como $\mathcal{TOY}(\mathcal{FD}g)$ e ILOG OPL, aunque estos últimos solo para ETP-7.

En Golomb, la clasificación completa de las tres instancias es: (1) MiniZinc, (2) Gecode, (3) $\mathcal{TOY}(\mathcal{FD}g)$, (4) $\mathcal{TOY}(\mathcal{FD}i)$, (5) ILOG Solver, (6) ILOG OPL, (7) SICStus Prolog, (8) PAKCS, (9) $\mathcal{TOY}(\mathcal{FD}s)$ y (10) SWI-Prolog. Por lo tanto, se puede afirmar que Golomb es un problema estable, ya que el orden obtenido para las tres instancias es fijo. Por otra parte, el peor rendimiento de cada sistema (con respecto al sistema más rápido, MiniZinc) aumenta a medida que las instancias aumentan. Este peor rendimiento con respecto a MiniZinc es respectivamente de 2,0-2,5, 4,0-5,5 y 7,0-9,5 veces más lento para los restantes sistemas basados en los resolutores de Gecode, ILOG Solver y SICStus `clpfd`. Sin embargo, si se compara al sistema mejor posicionado basado en el resolutor Gecode, ILOG Solver y SICStus `clpfd` (y, asimismo, a los tres sistemas posicionados como segundos basados en estos resolutores y a los tres sistemas posicionados como terceros, respectivamente) entonces el peor rendimiento de los sistemas ILOG Solver y SICStus `clpfd` (con respecto a los sistemas Gecode) disminuye en aproximadamente un 50%-60%.

En ETP-21 (la única instancia *js* de ETP), el orden completo es: (1) Gecode, (2) $\mathcal{TOY}(\mathcal{FD}g)$, (3) MiniZinc, (4) ILOG Solver, (5) $\mathcal{TOY}(\mathcal{FD}i)$, (6) ILOG OPL, (7) SICStus Prolog, (8) PAKCS, (9) $\mathcal{TOY}(\mathcal{FD}s)$ y (10) SWI-Prolog. El peor rendimiento (con respecto al sistema más rápido, Gecode) de los restantes sistemas Gecode, ILOG Solver y SICStus `clpfd` es de 1,01-1,53, 1,89-2,52 y 3,85-6,83 veces más lento, respectivamente. En este caso, si se compara los sistemas posicionados como primeros, como segundos y como terceros (respectivamente) basados en estos resolutores, entonces el rendimiento se modifica, convirtiéndose en un 15% mejor o peor (dependiendo de si los sistemas escogidos ocupan la segunda o tercera posición).

Considerando a las tres instancias de ETP, se puede afirmar que este es un problema menos estable, ya que el orden obtenido por cada sistema depende de la instancia concreta que se esté ejecutando. El peor rendimiento del sistema con orden

2 (respecivamente 3, 4 y así sucesivamente) con respecto a Gecode (que mantiene la primera posición en las tres instancias) disminuye a medida que las instancias aumentan. Sin embargo, como este orden entre sistemas varía para las distintas instancias, se requiere una segunda comparativa donde se compare a cada sistema concreto *s* con Gecode. Esta segunda comparación no permite obtener conclusiones generales: dependiendo del sistema elegido, su peor rendimiento con respecto a Gecode disminuye, aumenta, e incluso disminuye para despues aumentar, a medida que las instancias aumentan. Por otra parte, el orden se vuelve casi estable cuando se compara a los sistemas (basados en los resolutores de Gecode, ILOG Solver y SICStus `clpfd`) posicionados como primeros, segundos y terceros, respectivamente. Sin embargo, en este caso la sobrecarga inherente de cada paradigma juega un papel clave en la resolución de ETP-7, modificando el orden de rendimiento en la resolución, que ahora pasa a ser (1) Gecode, (2) SICStus y (3) ILOG Solver. Asimismo, en ETP-7, el peor rendimiento de los sistemas ILOG Solver y SICStus `clpfd` posicionados como segundos y terceros (respectivamente) disminuye en un 40%-80% con respecto al peor rendimiento de los sistemas ILOG Solver y SICStus `clpfd` posicionados como primeros. En ETP-15, la sobrecarga de cada paradigma no modifica el orden de rendimiento (1) Gecode, (2) ILOG Solver y (3) SICStus. Sin embargo, para esta instancia no pueden extraerse conclusiones generales acerca de cómo varía el rendimiento cuando se comparan a los sistemas posicionados como segundos y terceros (con respecto a los posicionados como primeros).

Finalmente, los resultados de los tres sistemas basados en Gecode, ILOG Solver y SICStus `clpfd` se consideran de manera independiente. Para cada uno de estos subconjuntos, la hipótesis inicial del análisis es que, en los modelos nativos (C++ CP($\mathcal{FD}$) Gecode, C++ CP($\mathcal{FD}$) ILOG Solver y CLP($\mathcal{FD}$) SICStus) la red de restricciones del COP se genera explícitamente, mientras que en MiniZinc, $\mathcal{TOY}(\mathcal{FD}g)$, ILOG OPL, $\mathcal{TOY}(\mathcal{FD}i)$, PAKCS y $\mathcal{TOY}(\mathcal{FD}s)$ esta red de restricciones se genera implícitamente, ya que depende de una compilación a la entrada aceptada por el resolutor en cuestión.

En cuanto a los sistemas basados en el resolutor de Gecode, el orden para las instancias de Golomb es: (1) MiniZinc, (2) Gecode y (3) $\mathcal{TOY}(\mathcal{FD}g)$. Sin embargo, para las instancias de ETP este orden es: (1) Gecode, (2) $\mathcal{TOY}(\mathcal{FD}g)$ y (3) MiniZinc. En la comparación entre Gecode y $\mathcal{TOY}(\mathcal{FD}g)$, el primero es siempre más rápido que el segundo. Sin embargo, mientras que, para las instancias *df*, el peor rendimiento de $\mathcal{TOY}(\mathcal{FD}g)$ es bastante grande, para las instancias *js* el rendimiento de $\mathcal{TOY}(\mathcal{FD}g)$ es prácticamente idéntico al de Gecode. La interfaz de $\mathcal{TOY}(\mathcal{FD})$ a la API de Gecode impone (al formular a los COP de Golomb y ETP) la misma red de restricciones que la impuesta por los modelos nativos C++ de Gecode. MiniZinc también impone la misma red de restricciones para el COP de Golomb, pero para ETP impone una red de restricciones 2,5 veces más grande que la impuesta por el modelo nativo Gecode. Sin embargo, el factor que está determinando la obtención de un mayor o menor rendimiento es la cantidad de restricciones que están siendo propagadas, así como la eficiencia del contructor de copia `copy()` de la clase `Space`. En cuanto a las propagaciones, se

ha comprobado que $\mathcal{TOY}(\mathcal{FD}g)$ efectúa prácticamente las mismas que Gecode (entre 1,01 y 1,02 veces más propagaciones). En el caso de MiniZinc, este efectúa solo un 0,30 de las propagaciones de Gecode para Golomb, siendo así 2,56 veces más rápido. Para ETP, MiniZinc efectúa de 1,33 a 1,47 veces más propagaciones (para los distintos equipos) que Gecode, siendo así de 1,53 a 1,57 veces más lento. En cuanto al método copy, se ha comprobado que el modelo Gecode que utiliza vectores bidimensionales (para modelar a tt y $trans\_tt$) obtiene un rendimiento de 1,22 a 1,31 veces más lento (para los distintos equipos) que el modelo Gecode que utiliza vectores unidimensionales. Además, esta diferencia del modelo *bidimensional* proviene directamente del peor rendimiento de su método `copy()` (con respecto al del modelo *unidimensional*) durante la exploración de búsqueda. En el caso de $\mathcal{TOY}(\mathcal{FD}g)$, el tiempo empleado en ejecutar su método `copy()` durante la exploración de búsqueda es prácticamente idéntico al del modelo Gecode *unidimensional*. Finalmente, se ha comprobado que tanto Gecode como $\mathcal{TOY}(\mathcal{FD}g)$ conducen a la exploración del mismo árbol de búsqueda (exactamente con la misma poda de los dominios de las variables en cada nodo del árbol). Sin embargo, se ha utilizado una restricción adicional para comprobar que, incluso alcanzando la misma poda de dominios, el orden y la cantidad de restricciones propagadas en Gecode y en $\mathcal{TOY}(\mathcal{FD}g)$ difieren.

En cuanto a los sistemas basados en el resolutor de ILOG Solver, el orden para las instancias de Golomb es: (1) $\mathcal{TOY}(\mathcal{FD}i)$, (2) ILOG Solver e (3) ILOG OPL. En ETP, mientras que el orden de las instancias *df* es (1) ILOG Solver, (2) ILOG OPL y (3) $\mathcal{TOY}(\mathcal{FD}i)$, el orden para la instancia *js* de ETP-21 es (1) ILOG Solver, (2) $\mathcal{TOY}(\mathcal{FD}i)$ y (3) ILOG OPL. Esto revela dos comportamientos diferentes por parte de $\mathcal{TOY}(\mathcal{FD}i)$: por un lado, su rendimiento se ve claramente penalizado en las instancias *df*, modificando el orden obtenido con respecto al de la instancia *js*. En este sentido, mientras que el peor rendimiento obtenido para las instancias *df* es muy grande, para la instancia *js* este se reduce a entre 1,06 y 1,33. Desafortunadamente, ni la API de ILOG OPL ni el de ILOG Solver proporcionan un método para calcular la cantidad de restricciones propagadas durante la exploración de búsqueda, lo que supone un inconveniente para el análisis de estos sistemas. En este contexto, se observa que los tres sistemas imponen sobre el almacén redes de restricciones diferentes y que, cuanto menor es esta red, mayor es el rendimiento del resolutor al efectuar la búsqueda. Respecto a estas redes, resulta interesante señalar que la diferencia de variables y restricciones impuestas en ILOG Solver y $\mathcal{TOY}(\mathcal{FD}i)$ es dependiente del COP: mientras que $\mathcal{TOY}(\mathcal{FD}i)$ utiliza menos variables y restricciones que ILOG Solver para G-11, para ETP-21 utiliza más. Finalmente, las ejecuciones de ILOG Solver y $\mathcal{TOY}(\mathcal{FD}i)$ han sido monitorizadas, identificando que hay casos en los que la interfaz de $\mathcal{TOY}(\mathcal{FD}i)$ permite ahorrar variables y restricciones con respecto al modelo nativo C++ de ILOG Solver, y viceversa.

En cuanto a los sistemas basados en el resolutor de SICStus `clpfd`, el orden para las instancias de Golomb y ETP es: (1) SICStus Prolog, (2) PAKCS y (3) $\mathcal{TOY}(\mathcal{FD}s)$. El peor rendimiento de PAKCS y $\mathcal{TOY}(\mathcal{FD}s)$ con respecto a SICStus es mayor para las

instancias *df* debido a la mayor sobrecarga del estrechamiento perezoso con respecto a la resolución SLD. Sin embargo, tampoco en las instancias *js* coincide el tiempo de resolución de SICStus, PAKCS y $\mathcal{TOY}(\mathcal{FD}s)$, revelando que hay un desajuste entre los modelos `clpfd` generados por los tres sistemas. En este contexto, independientemente del COP resuelto, PAKCS oscila entre 1,05 y 1,07 veces más lento que SICStus (en unos resultados similares a los obtenidos por $\mathcal{TOY}(\mathcal{FD}g)$ con respecto a Gecode). Sin embargo, el peor rendimiento de $\mathcal{TOY}(\mathcal{FD}s)$ con respecto a SICStus es dependiente del problema: mientras que, para Golomb, este oscila entre 1,07 y 1,11, para ETP- 21 aumenta notablemente, hasta 1,78. Esta diferencia de rendimiento entre Golomb y ETP se hace aún más clara cuando se compara con PAKCS. Mientras que el peor rendimiento de $\mathcal{TOY}(\mathcal{FD}s)$ con respecto a PAKCS decrece a medida que las instancias de Golomb aumentan, este aumenta a medida que las instancias de ETP aumentan. Se ha presentado un ejemplo sencillo en el que se imponía una única restricción $\mathcal{FD}$, pero desde dos escenarios diferentes. Se ha comprobado que `clpfd` realiza dos traducciones diferentes para dicha restricción, dando lugar a dos *indexicals* semánticamente equivalentes, pero basados en diferentes niveles de consistencia en sus algoritmos de propagación. Además, se ha comprobado que este nivel de propagación no es configurable en SICStus. Por otra parte, aunque ambos escenarios dan lugar a diferentes podas del dominio de las variables, esto no se refleja en las estadísticas de búsqueda mostradas, donde la única diferencia observada se refiere a la cantidad de veces que se ejecuta el algoritmo de filtrado. Obviamente, este es el factor que está determinando el tiempo de resolución de cada ejecución. Sin embargo, estos resultados no son relevantes por si mismos, ya que sería necesario conocer el tiempo que emplea cada algoritmo en ejecutarse y (en ejemplos con varios *indexicals*) el número de veces que se ejecuta cada uno de estos algoritmos. Desafortunadamente, SICStus no proporciona esta información. Finalmente, el predicado `call_residue` se ha utilizado para mostrar el almacén de restricciones de SICStus y $\mathcal{TOY}(\mathcal{FD}s)$ antes de iniciar la exploración de búsqueda para una instancia mínima G-5. Se ha comprobado que el conjunto de *indexicals* generado en ambos modelos difiere ligeramente, dando lugar a diferentes niveles de consistencia en sus algoritmos de propagación.

5 Conclusiones y trabajo futuro

En esta sección se presentan las principales conclusiones de la investigación realizada en esta tesis, que ha sido dividida en tres partes: la mejora del rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD})$, dos aplicaciones reales de $\mathcal{TOY}(\mathcal{FD})$, y un posicionamiento de $\mathcal{TOY}(\mathcal{FD})$ con respecto a otros sistemas CP($\mathcal{FD}$) de vanguardia para el modelado y resolución de dos COP. Las secciones 5.1, 5.2 y 5.3 presentan las conclusiones de cada una de estas partes, y la sección 5.4 presenta posibles vías de trabajo futuro para cada una de ellas.

5.1 Mejora del rendimiento de $\mathcal{TOY}(\mathcal{FD})$

Se ha presentado un esquema para integrar resolutores de restricciones C++ CP($\mathcal{FD}$) en $\mathcal{TOY}(\mathcal{FD})$ (en un contexto fácilmente adaptable a otros sistemas CLP($\mathcal{FD}$) o CFLP($\mathcal{FD}$) implementados en Prolog). Se han descrito las dificultades adicionales surgidas por la falta de correspondencia entre las variables, restricciones y tipos soportados por el sistema y resolutor. Además, se ha descrito la adaptación de un resolutor C++ CP($\mathcal{FD}$) a los requisitos de un sistema CFLP($\mathcal{FD}$), como son el razonamiento por modelos, el uso de múltiples estrategias de búsqueda (intercaladas con el almacenamiento de nuevas restricciones) y un modo de propagación tanto incremental como por lotes. El esquema ha resultado ser ciertamente génerico, puesto que ha permitido integrar dos resolutores diferentes (como son Gecode e ILOG Solver, dando lugar a las nuevas versiones del sistema $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$, respectivamente) siguiendo simplemente los pasos que se describen en el esquema.

El rendimiento de resolución de las tres veriones $\mathcal{TOY}(\mathcal{FD})$ ha sido analizado mediante el uso de tres CSP clásicos (series mágicas, reinas y números de Langford) y un COP clásico (reglas de Golomb). Estos problemas abarcan todo el repertorio de restricciones $\mathcal{FD}$ soportadas por $\mathcal{TOY}(\mathcal{FD})$. Además, mediante el uso de tres instancias por problema (resueltas en órdenes de magnitud de décimas de segundo, segundos y minutos, respectivamente) se ha comparado el rendimiento de $\mathcal{TOY}(\mathcal{FD})$ a medida que la complejidad computacional del problema aumenta.

Se ha comprobado que $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$ mejoran el rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD}s)$, pero la mejora lograda (que va desde 1,15 a 3,57 veces más rápido) es dependiente del problema e instancia concreta a resolver. Para los problemas de series mágicas y Golomb, la mejora de $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a $\mathcal{TOY}(\mathcal{FD}s)$ permanece estable a medida que las instancias aumentan. Para los casos de las reinas y de Langford, la mejora alcanzada aumenta desde la instancia resuelta en décimas de segundo a las instancias resueltas en segundos y minutos. En cualquier caso, el modo de propagación no juega un papel clave en el tiempo de resolución, ya que, aunque el modo por lotes es más rápido que el incremental para todos los casos,

las diferencias obtenidas son muy pequeñas (aproximadamente un orden de magnitud menor que el tiempo de resolución de la instancia, o incluso menos).

Por otra parte, existe una clara correlación entre el porcentaje de tiempo de resolución del problema que se dedica a la búsqueda y la mejora de rendimiento alcanzada. Así, mientras el porcentaje se mantiene estable en los problemas de series mágicas y Golomb, este aumenta claramente desde las instancias de las reinas y Langford resueltas en décimas de segundo a las instancias resueltas en segundos y minutos. Esto convierte a la resolución de $\mathcal{TOY}(\mathcal{FD})$ en una tarea completamente dependiente de CP($\mathcal{FD}$). Es decir, el tiempo que cada versión de $\mathcal{TOY}(\mathcal{FD})$ emplea para la resolución de un problema viene determinado unívocamente por el rendimiento ofrecido por su resolutor $\mathcal{FD}$ para acometer una búsqueda (mediante la propagación de restricciones básicas y globales).

En este sentido, el otro enfoque adecuado para mejorar el rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$ se centra en la estrategia de búsqueda aplicada; específicamente, en sustituir una estrategia básica por una nueva estrategia *ad hoc* que explote el conocimiento acerca de la estructura del problema y sus soluciones. Es decir, manteniendo el mismo resolutor para acometer la búsqueda, modificar el modelo $\mathcal{TOY}(\mathcal{FD})$ para especificar una estrategia de búsqueda que requiera una menor exploración para encontrar las soluciones al problema. La motivación de esta aproximación (fácilmente adaptable a otros sistemas CLP($\mathcal{FD}$) o CFLP($\mathcal{FD}$) implementados en Prolog e integrando resolutores C++ CP($\mathcal{FD}$)) se basa en aprovechar tanto la gran expresividad de $\mathcal{TOY}(\mathcal{FD})$ para especificar estrategias de búsqueda, como la alta eficiencia de Gecode e ILOG Solver para acometer estas búsquedas.

Por un lado se han presentado ocho nuevas primitivas de búsqueda que incluyen algunos conceptos novedosos (no directamente disponibles en Gecode ni en ILOG Solver): una exploración exhaustiva en anchura sobre los primeros niveles del árbol (ordenando posteriormente los nodos satisfactibles por un criterio específico); una fragmentación del dominio de las variables, acotando cada una de ellas a un subconjunto de los valores de su dominio (en lugar de asignándolas directamente a un valor concreto); y la aplicación de estas estrategias de asignación o de fragmentación únicamente a un subconjunto de las variables involucradas. Por otro lado, también se ha remarcado la expresividad y flexibilidad de $\mathcal{TOY}(\mathcal{FD})$ para especificar algunos criterios de búsqueda, así como que el uso de un razonamiento por modelos permite aplicar diferentes estrategias de búsqueda (generando diferentes escenarios) a la resolución de un problema.

Las nuevas versiones de $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$ han ampliado las bibliotecas de Gecode e ILOG Solver utilizando sus diferentes mecanismos de búsqueda subyacentes (`Search Engine`, `Brancher` y *recomputación híbrida* en Gecode; `IloGoal`, constructor de `IloGoal` y pila de ejecución de `IloGoal` en ILOG Solver). En primer lugar, se ha presentado un esquema abstracto que enumera los requisitos necesarios para integrar estas estrategias de búsqueda en $\mathcal{TOY}(\mathcal{FD})$ (y que ha sido instanciado

posteriormente a Gecode e ILOG Solver). Además, se ha analizado el impacto en la arquitectura del sistema de la implementación de estas primitivas. Esto ha revelado que la especificación de criterios de búsqueda directamente en $\mathcal{TOY}(\mathcal{FD})$ genera una sobrecarga computacional debido a la interacción recursiva entre las capas Prolog y C++ de dicha arquitectura.

Los modelos $\mathcal{TOY}(\mathcal{FD})$ para las series mágicas, reinas, números de Langford y Golomb se han revisado, discutiendo la estructura de las soluciones de cada problema y analizando cómo el uso de las nuevas estrategias de búsqueda propuestas permite reducir la exploración para encontrar estas soluciones. Se han desarrollado nuevos modelos $\mathcal{TOY}(\mathcal{FD})$, equivalentes a los anteriores, pero con una nueva estrategia de búsqueda *ad hoc*. Se ha comprobado que estas estrategias mejoran el rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a los modelos originales (en un rango desde 1,05 veces más rápido a más de 1000). En particular, mientras que para los problemas de las reinas y de Langford el mayor rendimiento alcanzado claramente aumenta a medida que el tamaño de las instancias aumentan, para las series mágicas este mayor rendimiento se mantiene estable, y para Golomb disminuye. Por otra parte, el mayor rendimiento alcanzado por $\mathcal{TOY}(\mathcal{FD}g)$ con respecto a $\mathcal{TOY}(\mathcal{FD}i)$ aumenta con respecto al obtenido para los modelos originales, lo que revela que el enfoque de Gecode para ampliar su biblioteca con nuevas estrategias de búsqueda es más eficiente que el de ILOG Solver.

5.2 Aplicaciones reales de $\mathcal{TOY}(\mathcal{FD})$

Se han presentado dos aplicaciones industriales del sistema. La primera de ellas ha consistido en un problema de asignación de trabajadores a turnos de trabajo (ETP) proveniente de la industria de las comunicaciones. A diferencia del conjunto de problemas de prueba utilizado en la primera parte de la investigación (series mágicas, reinas, números de Langford y Golomb), este ETP explota tanto la gran expresividad de $\mathcal{TOY}(\mathcal{FD})$ como su mayor rendimiento de resolución recién alcanzado en las versiones $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$.

La aproximación para abordar el COP ha sido presentada en detalle, siendo esta totalmente paramétrica, no monolítica e incluyendo componentes independientes de CP($\mathcal{FD}$). Se han propuesto tres instancias (con tiempos de resolución de órdenes de magnitud de décimas de segundo, segundos y minutos, respectivamente), para comparar las conclusiones de ETP con las obtenidas para el conjunto de problemas de prueba. Se ha comprobado que ETP se comporta de manera similar a los problemas de las reinas y de Langford. En concreto la mejora de rendimiento de $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a $\mathcal{TOY}(\mathcal{FD}s)$ aumenta a medida que el tamaño de las instancias aumenta. También aumenta el porcentaje de tiempo de resolución dedicado a la búsqueda y el impacto de la estrategia de búsqueda *ad hoc* aplicada.

Sin embargo, los resultados para ETP son más dependientes de la instancia concreta que se ejecuta que los resultados de los problemas de las reinas y Langford: $\mathcal{TOY}(\mathcal{FD}g)$ todavía supera a $\mathcal{TOY}(\mathcal{FD}s)$, pero el rango de mejora alcanzada es más amplio que para las reinas y de Langford. $\mathcal{TOY}(\mathcal{FD}i)$ se comporta peor, igual o mejor que $\mathcal{TOY}(\mathcal{FD}s)$ para las tres instancias de ETP, respectivamente. En cualquier caso, el porcentaje de exploración de búsqueda de $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$ para las instancias de ETP resueltas en décimas de segundo y segundos es mucho menor que el de $\mathcal{TOY}(\mathcal{FD}s)$, e incluso que los de $\mathcal{TOY}(\mathcal{FD}g)$ y $\mathcal{TOY}(\mathcal{FD}i)$ para las instancias de las reinas y de Langford. En cuanto a la propagación por lotes, en $\mathcal{TOY}(\mathcal{FD}i)$ esta es mucho más eficiente que el modo incremental para las instancias resueltas en décimas de segundo y segundos, y en $\mathcal{TOY}(\mathcal{FD}s)$ para las instancias resueltas en segundos y minutos. Finalmente, el impacto de aplicar una estrategia de búsqueda *ad hoc* es menor en ETP que en los casos de las reinas y Langford. Además, el mejor rendimiento de $\mathcal{TOY}(\mathcal{FD}g)$ con respecto a $\mathcal{TOY}(\mathcal{FD}i)$ no aumenta a medida que las instancias aumentan.

Una segunda aplicación real ha consistido en un análisis empírico acerca de la complejidad computacional del problema de asignación de elementos unidimensionales a contenedores (BPP). Este análisis ha incluido un conjunto de instancias generadas paramétricamente mediante configuraciones del modelo estadístico de Weibull. El conjunto de instancias se ha resuelto aplicando dos modelos CP($\mathcal{FD}$) equivalentes (Gecode y $\mathcal{TOY}(\mathcal{FD}g)$) y cuatro heurísticas (MAXREST, FIRSTFIT, BESTFIT y NEXTFIT). Las conclusiones obtenidas son interesantes para el futuro desarrollo de resolutores especializados que aborden a diferentes COP de configuración (que pueden verse como generalizaciones del BPP) provenientes de la optimización en la industria de centros de datos.

Se ha analizado la flexibilidad de Weibull, comprobándose que permite generar una gran variedad de distribuciones en el tamaño de los elementos mediante el uso de diferentes combinaciones de los parámetros (k, λ). Se han modelado (con gran precisión) un conjunto de instancias BPP reales usando Weibull (con gráficos de máxima verosimilitud y Q-Q, para mostrar visualmente la calidad del ajuste, así como con tests estadísticos KS y χ^2, para probar este ajuste de un modo más riguroso). Se ha generado un conjunto de 19.900 instancias mediante 199 categorías o combinaciones diferentes de (k, λ), con 100 instancias para cada una de estas. Específicamente, el parámetro λ ha sido fijado a 1000 (expandiendo el tamaño de los elementos de la distribución hasta tres órdenes de maginitud), y el parámetro k ha oscilado en el rango $[0,1,\ 0,2,\ \ldots,\ 19,9]$. Una vez fijado el conjunto de instancias, se han analizado 11 capacidades de contenedor C diferentes, oscilando su tamaño entre 1,0 y 2,0 (con incrementos de 0,1) veces el tamaño del elemento más grande de la instancia a resolver.

El análisis de los resultados de CP($\mathcal{FD}$) ha revelado que, para todo valor de C, existen categorías (k, λ) para los que esta aproximación encuentra dificil resolver buena parte del conjunto de las 100 instancias. Más especificamente, la complejidad que una

determinada categoría resulta para CP($\mathcal{FD}$) se clasifica en 5 grupos, según el porcentaje de instancias resueltas: 80%-100%, 60%-80%, 40%-60%, 20%-40% y 0%-20%. En este sentido se ha comprobado que incrementar C también aumenta la complejidad computacional del problema, de manera que la cantidad de instancias resueltas disminuye según aumenta C. Con $C = 1,0$, 178 categorías pertenecen al grupo 80%-100% y 21 al grupo 60%-80%. Con $C = 1,5$, 150 categorías pertenecen al grupo 80%-100%, 17 al grupo 60%-80%, 18 al grupo 40%-60% y 14 al grupo 20%-40%. Con $C = 2,0$, 87 categorías pertenecen al grupo 80%-100%, 20 al grupo 60%-80%, 12 al grupo 40%-60%, 18 al grupo 20%-40% y 62 al grupo 0%-20% (incluyendo 12 categorías para los que el porcentaje se reduce prácticamente al 0%). Además, existe una correlación entre el porcentaje de instancias resueltas y el tiempo dedicado a la búsqueda de la solución óptima.

Por otra parte, a medida que k aumenta, también lo hace la cantidad de contenedores utilizados en la solución óptima. Por ello, como todas las instancias contienen 100 elementos, se puede decir que hay una correspondencia entre cada categoría (perteneciente a uno de los grupos) y el ratio de elementos asignados a cada contenedor. Sin embargo, fijando k, se comprueba que este ratio varía a medida que varía C. Por un lado, se puede concluir que, si la categoría requiere un ratio de menos de 1,60 o más de 4,00 elementos por contenedor, entonces la categoría va a ser clasificada dentro del grupo 80%-100%. Por otro lado, para el resto de grupos no se pueden extraer conclusiones generales con respecto a las diferentes configuraciones de C. En concreto para $C = 2,0$ y $k = [10,0,\ldots,19,9]$ hay categorías de todos los grupos, todas ellas con una misma cantidad media de $2,0$ elementos por contenedor.

Las heurísticas resuelven el conjunto de instancias en una cantidad insignificante de tiempo, por lo que, para ellas, el análisis se ha centrado en la desviación media de MAXREST (con resultados muy similares a los de FIRSTFIT y BESTFIT) y NEXTFIT con respecto al número óptimo de contenedores computado por CP($\mathcal{FD}$). En este sentido, la desviación media de MAXREST es un orden de magnitud menor (entre 0,0 y 1,4 contenedores) que la de NEXTFIT (entre 0,0 y 20,0 contenedores). En MAXREST, a medida que C aumenta, también aumenta la diferencia media con las soluciones óptimas computadas por CP($\mathcal{FD}$). Además, existe una clara correlación entre las categorías pertenecientes a cada uno de los grupos y la diferencia en el número de contenedores computados por la heurística y CP($\mathcal{FD}$). Con NEXTFIT, se ha comprobado que hay una frontera en $C = 1,5$. Así, mientras que, desde $C = 1,0$ a $C = 1,5$, la diferencia de contenedores entre la heurística y CP($\mathcal{FD}$) aumenta, desde $C = 1,6$ a $C = 2,0$ esta diferencia disminuye. Desafortunadamente, no hay ninguna correlación entre las categorías y la diferencia obtenida. En general, mientras que la heurística MAXREST (y, por tanto, también FIRSTFIT y BESTFIT) representa una muy buena alternativa a CP($\mathcal{FD}$) (ya que obtiene muy buenas aproximaciones a la solución óptima en una cantidad casi insignificante de tiempo), la diferencia obtenida por NEXTFIT no la hace una alternativa tan interesante.

5.3 $\mathcal{TOY}(\mathcal{FD})$ comparado con otros sistemas CP($\mathcal{FD}$)

Se ha realizado una comparativa acerca del modelado y resolución de los COP de Golomb y ETP entre varios sistemas CP($\mathcal{FD}$) de vanguardia. En concreto se han seleccionado los sistemas CP($\mathcal{FD}$) algebraicos MiniZinc e ILOG OPL, los sistemas C++ CP($\mathcal{FD}$) Gecode e ILOG Solver, los sistemas CLP($\mathcal{FD}$) SICStus Prolog y SWI-Prolog, así como los sistemas CFLP($\mathcal{FD}$) PAKCS y $\mathcal{TOY}(\mathcal{FD})$ (este último con sus tres versiones diferentes $\mathcal{TOY}(\mathcal{FD}g)$, $\mathcal{TOY}(\mathcal{FD}i)$ y $\mathcal{TOY}(\mathcal{FD}s)$).

Desde el punto de vista del lenguaje $\mathcal{TOY}(\mathcal{FD})$, se concluye que su uso resulta una alternativa atractiva para el modelado de estos COP, ya que proporciona una serie de ventajas: abstrae la noción del resolutor de restricciones, gestionando el uso de diferentes resolutores y el almacenamiento de las restricciones en ellos de un modo transparente al usuario. Proporciona un libre acceso a las variables involucradas en el COP. Permite modelar los problemas en un solo archivo, capturando la arquitectura de varias etapas del algoritmo p_tt mediante la simple enumeración en orden de estas. Soporta el uso de estructuras de datos dinámicas, y proporciona un acceso sencillo a sus elementos. Permite ahorrar aquellas variables $\mathcal{FD}$ del problema que se puedan unificar directamente a otras variables o a valores enteros, efectuando estas unificaciones antes de imponer ninguna restricción $\mathcal{FD}$ sobre la variable en cuestión. Proporciona primitivas para los modos de propagación incremental y por lotes, lo que permite aplicar diferentes modos de propagación a diferentes partes del programa. En términos generales, proporciona un lenguaje declarativo de propósito general, incluyendo características de modelado muy expresivas, como son las funciones indeterministas, el uso de tipos, el orden superior, la evaluación perezosa, el encaje de patrones o la aplicación parcial. Todo ello permite al usuario escribir formulaciones más claras y concisas. En concreto se ha comprobado que únicamente los sistemas CP($\mathcal{FD}$) algebraicos requieren una menor cantidad de líneas de código para el modelado de los problemas.

Desde el punto de vista de $\mathcal{TOY}(\mathcal{FD}g)$, se concluye que su uso es altamente recomendable para la resolución de estos COP. Se ha comprobado que la interfaz de $\mathcal{TOY}(\mathcal{FD})$ a la API de Gecode permite generar un modelo Gecode prácticamente idéntico al modelo C++ CP($\mathcal{FD}$) (logrando así el mismo rendimiento que al programar un modelo nativo Gecode). En este contexto, como tanto Gecode como $\mathcal{TOY}(\mathcal{FD}g)$ contienen prácticamente la misma red de restricciones, el tiempo que dedican a la búsqueda es prácticamente idéntico (con una mínima sobrecarga de un 1%-2% en $\mathcal{TOY}(\mathcal{FD}g)$, debida a la propagación de unas pocas restricciones más durante la búsqueda). Para las instancias *df*, el rendimiento de $\mathcal{TOY}(\mathcal{FD}g)$ sigue siendo peor que el de Gecode, ya que este es penalizado por las sobrecargas debidas al estrechamiento perezoso y la comunicación con el resolutor externo. Pero para las instancias *js*, donde la búsqueda es el único factor determinante para el tiempo de resolución, $\mathcal{TOY}(\mathcal{FD}g)$ prácticamente alcanza el rendimiento de Gecode, convirtiéndose en el segundo sis-

tema más rápido para la resolución de Golomb, y en el sistema más rápido para la resolución de ETP.

Desde el punto de vista de $\mathcal{TOY}(\mathcal{FD}i)$, se concluye que su uso es recomendable para la resolución de estos COP, aunque su rendimiento es claramente dependiente de la formulación del problema. La interfaz de $\mathcal{TOY}(\mathcal{FD})$ necesita un mayor código de pegamento para coordinar la gestión de los objetos de ILOG Concert e ILOG Solver, lo que implica una sobrecarga adicional relevante. Mientras que esta sobrecarga es crucial para las instancias *df* (provocando un rendimiento mucho peor que el del modelo nativo ILOG Solver) este se vuelve casi insignificante para las instancias *js*. Además, este código adicional produce un desajuste entre la red de restricciones generada por $\mathcal{TOY}(\mathcal{FD}i)$ y la generada por el modelo ILOG Solver nativo (tanto en la cantidad de variables $\mathcal{FD}$ como de restricciones $\mathcal{FD}$ utilizadas). Se ha comprobado que, mientras que hay casos en las que $\mathcal{TOY}(\mathcal{FD}i)$ ahorra variables y restricciones con respecto a ILOG Solver, en otras ocasiones es justo al revés. Por lo tanto, la red de restricciones generada por $\mathcal{TOY}(\mathcal{FD}i)$ e ILOG Solver es totalmente dependiente de la formulación del problema a tratar (más específicamente, de la cantidad de estas situaciones de desajuste que se producen en dicha formulación). En este contexto, el rendimiento de $\mathcal{TOY}(\mathcal{FD}i)$ es de un 6% a un 11% más rápido que el de ILOG Solver para las instancias de Golomb (convirtiéndose así en prácticamente el cuarto sistema más rápido), pero un 17% más lento para ETP-21 (convirtiéndose así en prácticamente el quinto sistema más rápido).

Desde el punto de vista de $\mathcal{TOY}(\mathcal{FD}s)$, se concluye que su uso es recomendable para la resolución del COP puramente CP($\mathcal{FD}$) de Golomb, pero no tan aconsejable para la resolución del COP industrial de ETP. Respecto a Golomb, el rendimiento de $\mathcal{TOY}(\mathcal{FD}s)$ sufre una sobrecarga de un 7% a un 10% con respecto al modelo nativo SICStus, y alcanza un rendimiento prácticamente idéntico al de PAKCS. Sin embargo, respecto al ETP, la sobrecarga de $\mathcal{TOY}(\mathcal{FD}s)$ oscila entre un 70% y un 80% con respecto al modelo nativo SICStus, siendo por tanto mucho menos competitivo con respecto a este (y también con respecto a PAKCS). Se ha comprobado que este desajuste en el rendimiento es debido a la traducción entre restricciones e *indexicals* realizada por `clpfd`. Esta difiere entre las ejecuciones de SICStus y $\mathcal{TOY}(\mathcal{FD}s)$, dando lugar a diferentes redes de restricciones (es decir, conjuntos de *indexicals* de idéntica cardinalidad, pero basados en diferentes algoritmos de propagación).

En resumen, más allá del interés general de comparar estos sistemas CP($\mathcal{FD}$) de vanguardia, los resultados han mostrado a $\mathcal{TOY}(\mathcal{FD})$ como un sistema competitivo con respecto a los demás sistemas involucrados en el análisis, tanto para el modelado como para la resolución de estos COP. Esto es relevante, pues fomenta tanto el uso del sistema como del propio paradigma CFLP($\mathcal{FD}$) al que pertenece.

5.4 Trabajo futuro

El esquema genérico propuesto en la sección 2.1 se puede reutilizar para integrar nuevos resolutores C++ CP($\mathcal{FD}$) que proporcionen un mayor rendimiento que los de Gecode e ILOG Solver. Además de esto, la compleja naturaleza de muchos CSP y COP hace que su especificación contenga varios dominios de variables. Aunque Gecode e ILOG Solver soportan variables enteras, Booleanas, de conjuntos e incluso reales, en ocasiones la complejidad computacional de estos CSP y COP hace intratable la resolución del problema mediante la simple aplicación de una búsqueda exhaustiva. Un claro ejemplo de esto son los BPP generalizados provenientes de la optimización en la industria de centros de datos, donde se necesitan complejos planes dinámicos para su resolución. En estos casos se requiere la combinación de CP($\mathcal{FD}$) con otras técnicas. Por ejemplo, la *generación perezosa de claúsulas* [187] ha combinado recientemente CP($\mathcal{FD}$) con resolución SAT, y se ha comprobado su eficacia para resolver gran cantidad de problemas industriales para los que todavía no se conocía una solución óptima [174].

En $\mathcal{TOY}$ existe una versión del sistema que combina CP($\mathcal{FD}$) con técnicas de programación matemática [68] mediante la coordinación de los resolutores de restricciones SICStus `clpfd` y `clpr` (de ahora en adelante $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$). Este usa un resolutor específico $\mathcal{M}$, que actúa como mediador para la gestión de las restricciones puente `X #== Y` que aparecen en el cómputo de un objetivo. Cada una de estas restricciones solicita a la variable real Y tomar un valor equivalente al de la variable entera X. A partir de este punto, los resolutores `clpfd` y `clpr` trabajan de forma independiente, pero cualquier poda en X o Y es transmitida explícitamente de un resolutor a otro mediante proyecciones de la restricción puente [104]. En este sentido, el sistema $\mathcal{TOY}(\mathcal{FD}i)$ se puede extender para reproducir el esquema propuesto, con ILOG Solver reemplazando a SICStus `clpfd` e ILOG CPLEX [13] reemplazando a SICStus `clpr`. El mayor rendimiento de ILOG Solver e ILOG CPLEX con respecto a SICStus `clpfd` y `clpr` (respectivamente) permitiría mejorar el rendimiento de resolución de la nueva versión del sistema $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$ con respecto a la actual (del mismo modo que el mayor rendimiento de ILOG Solver con respecto a SICStus `clpfd` mejoró el rendimiento de resolución de $\mathcal{TOY}(\mathcal{FD}i)$ con respecto a $\mathcal{TOY}(\mathcal{FD}s)$). Además, tanto ILOG Solver como ILOG CPLEX hacen uso de la misma biblioteca de modelado ILOG Concert. Por ello, la nueva versión del sistema podría implementar al propio resolutor mediador $\mathcal{M}$ en ILOG Concert, en lugar de en la capa Prolog de la arquitectura del sistema, mejorando aún más el rendimiento de resolución. Como un aspecto negativo, hay que decir que ILOG CPLEX siempre proporciona una respuesta extensional del sistema de ecuaciones propuesto (asignando a las variables a un valor concreto), en lugar de ofrecer una respuesta intensional (simplificando el sistema), como hace SICStus `clpr`.

Las estrategias de búsqueda *ad hoc*, propuestas en la sección 2.2, se podrían aplicar a conjuntos de problemas de prueba CP($\mathcal{FD}$) clásicos. Se podrían analizar múltiples y

muy precisos escenarios de búsqueda, mediante el uso de técnicas de procesamiento como las propuestas para $\mathcal{TOY}(\mathcal{FD}g)$ en la sección 3.2 (basadas en funciones indeterministas). En particular, sería interesante analizar una combinación exhaustiva de la aplicación de una o varias estrategias de búsqueda (así como del subconjunto de variables etiquetadas o fragmentadas por cada una de estas). Los resultados obtenidos serían analizados con el fin de averiguar qué estrategias han dado lugar a una solución del problema, o, al menos, a un espacio de búsqueda mínimo que contenga una solución. Por otra parte, este análisis ayudaría a descubrir nuevos patrones acerca de la relación existente entre la estructura de un problema y la estrategia de búsqueda (o combinación de estrategias de búsqueda) a aplicar para resolverlo con éxito.

El modelo de ETP de la sección 3.1 es totalmente paramétrico, por lo que se podrían analizar múltiples nuevas instancias. Estas podrían variar en el número de días a planificar, número de equipos y de trabajadores por equipo, la periodicidad con que el trabajador adicional se puede seleccionar, el factor extra al que sus horas de trabajo se deben pagar, el número de diferentes tipos de jornadas de trabajo (y los turnos concretos solicitados en cada una de ellas), las ausencias de trabajadores y el nivel de homogeneidad requerido en la distribución de cada tipo de turno de trabajo entre los trabajadores de cada equipo. Por otra parte, el uso de múltiples equipos disjuntos es una característica particular de la formulación del problema, pero la posibilidad de utilizar un único equipo también se puede considerar (estableciendo simplemente que el parámetro concreto tome valor uno). Esto permitiría aumentar la aplicabilidad del modelo propuesto, reutilizándolo para otros muchos problemas industriales de asignación de recursos.

Las conclusiones del análisis empírico de BPP, presentado en la sección 3.2, constituyen una base para el futuro desarrollo de resolutores de restricciones especializados, que aborden instancias generalizadas provenientes de la optimización en la industria de centros de datos. Aplicando técnicas de máxima verosimilitud a instancias reales, es fácil determinar los parámetros de Weibull (k,λ) que permiten una mejor representación de la distribución de elementos de la instancia. Comparando estos parámetros (o categoría) obtenidos con los resultados del análisis empírico, sería sencillo determinar si es mejor hacer uso de CP($\mathcal{FD}$) o de las heurísticas para resolver el problema. Finalmente, la generación paramétrica de un conjunto de instancias, siguiendo un modelo estadístico, se podría extender a otros problemas importantes de asignación de recursos (como forma de estudiar instancias reales).

Por otra parte, como ya se indicó en la sección 1.4, existen múltiples aplicaciones reales de CP($\mathcal{FD}$), provenientes de una amplia gama de dominios de aplicación. Basándose únicamente en los artículos publicados en la pasada edición de CP'13, se puede encontrar ejemplos de planificación para: centrales eléctricas [26], terrenos para la agricultura [35], sistemas urbanos para compartir bicicletas [77], sistemas caóticos dinámicos [84], utilización de materiales en la industria del metal [117], comportamiento atómico en la industria química [129], asignación de redes inhalámbricas [143], asig-

nación de vehículos para la construcción [208] y mezcla de vino [196]. Asimismo, se pueden encontrar aplicaciones para la verificación de hardware [140] y generación de preguntas para juegos matemáticos [191]. Por lo tanto, sería interesante analizar la aplicación de $\mathcal{TOY}(\mathcal{FD})$ a algunos de estos problemas y dominios de aplicación.

En cuanto a la comparación de sistemas CP($\mathcal{FD}$) (como la realizada en la sección 4), se podrían considerar más problemas, tanto clásicos (como los existentes en el catálogo de CSPlib) como reales (como los que se acaban de mencionar, provenientes del CP'13). En el caso de ETP, siguiendo la propuesta de analizar múltiples nuevas instancias que exploten la parametrícidad del problema, se podrían añadir estas nuevas instancias a la comparativa de resolución. Además, se podrían considerar más sistemas CP($\mathcal{FD}$) de vanguardia, como por ejemplo Comet [60], Minion [81] o Numberjack [98] (para sistemas CP($\mathcal{FD}$) algebraicos), JaCoP [110], Choco [54] o YACS [206] (para sistemas Java CP($\mathcal{FD}$)), y GNU Prolog [5], Ciao Prolog [4], B-Prolog [31] o ECLiPSe [16] (para sistemas CLP($\mathcal{FD}$)).

Por otra parte, $\mathcal{TOY}(\mathcal{FD})$ podría compararse con respecto a la aplicación de otras técnicas diferentes de CP($\mathcal{FD}$), como la programación matemática y la resolución SAT. En primer lugar, el problema de ETP se podría modelar con ILOG CPLEX, desarrollando un modelo nativo C++ CP($\mathcal{R}$). Una comparación entre los modelos de ILOG Solver, $\mathcal{TOY}(\mathcal{FD}i)$ e ILOG CPLEX permitiría discutir las ventajas que CP($\mathcal{FD}$) proporciona para la formulación de un problema tan orientado a restricciones como lo es este. Además, un análisis para la resolución de múltiples instancias de ETP permitiría comparar el rendimiento de aplicar una búsqueda exhaustiva o un enfoque basado en el simplex [141]. En segundo lugar, la nueva versión de $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$ propuesta también se podría aplicar al problema. En este caso, una comparativa de modelado entre $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$ y el modelo nativo C++ CP($\mathcal{R}$) de ILOG CPLEX permitiría evaluar cómo la mayor abstracción de $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$ facilita la formulación del problema y comparar esta diferencia de abstracción con la existente entre ILOG Solver y $\mathcal{TOY}(\mathcal{FD})$. Siguiendo con $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$, una comparativa de su rendimiento de resolución con el de ILOG CPLEX permitiría analizar si la interfaz de $\mathcal{TOY}(\mathcal{FD}\&\mathcal{R})$ a ILOG CPLEX produce el mismo tipo de desajustes en la red de restricciones generadas que el producido entre $\mathcal{TOY}(\mathcal{FD})$ y los modelos nativos de ILOG solver (donde el desajuste en la red de restricciones generada era dependiente de la formulación del problema). Finalmente, se podría analizar una reformulación del problema ETP para su modelado en un resolutor SAT.