Brahim Hnich

Function Variables for Constraint Programming
Function Variables
for Constraint Programming

Brahim Hnich

Computer Science Division
Department of Information Science
Uppsala University

2003
Dissertation for the Degree of Doctor of Philosophy in Computer Science presented at Uppsala University in 2003

ABSTRACT


Quite often modelers with constraint programming (CP) use the same modelling patterns for different problems, possibly from different domains. This results in recurring idioms in constraint programs. Our approach can be seen as a three-step approach. First, we identify some of these recurring patterns in constraint programs. Second, we propose a general way of describing these patterns by introducing proper constructs that would cover a wide range of applications. Third, we propose automating the process of reproducing these idioms from these higher-level descriptions. The whole process can be seen as a way of encapsulating some of the expertise and knowledge often used by CP modelers and making it available in much simpler forms. Doing so, we are able to extend current CP languages with high-level abstractions that open doors for automation of some of the modelling processes.

In particular, we introduce function variables and allow the statement of constraints on these variables using function operations. A function variable is a decision variable that can take a value from a set of functions as opposed to an integer variable that ranges over integers, or a set variable that ranges over a set of sets. We show that a function variable can be mapped into different representations in terms of integer and set variables, and illustrate how to map constraints stated on a function variable into constraints on integer and set variables. As a result, a function model expressed using function variables opens doors to the automatic generation of alternate CP models. These alternate models either use a different variable representation, or have extra implied constraints, or employ different constraint formulation, or combine different models that are linked using channeling constraints. A number of heuristics are also developed that allow the comparison of different constraint formulations. Furthermore, we present an extensive theoretical comparison of models of injection problems supported by asymptotic and empirical studies. Finally, a practical modelling tool that is built based on a high-level language that allows function variables is presented and evaluated. The tool helps users explore different alternate CP models starting from a function model that is easier to develop, understand, and maintain.

Key words: Constraint satisfaction, constraint programming, high-level modelling, abstraction, reformulation, function variables.

Brahim Hnich, Division of Computer Science, Department of Information Science, Uppsala University, BOX 513 SE-751 20 Uppsala, Sweden

© Brahim Hnich 2003

ISBN 91–506–1650–1

Printed in Sweden by Uppsala University Tryck & Medier, Uppsala 2003

Distributor: Department of Information Science, Uppsala University
To my parents for their everlasting love, support, and encouragements!
Acknowledgments

This work has significantly benefited from the scientific feedback from Toby Walsh. This research has been supported by the department of Information Science, Uppsala University, and the Swedish Research Council, under Pierre Flener’s Research Grant number 221-99-369.

I also want to extend my gratitude to Andreas Hamfelt for being an excellent head of the department, amazingly supportive, and an inspiring manager. I am in debt to my wonderful colleague and dear friend Zeynep Kiziltan, with whom I very much enjoyed working. Thank you Zeynep for sharing with me so many happy moments as well as being a great support during the not very pleasant times. I am forever grateful that I have friends like Monica Tavanti and Timour Katchaounov, for I would never survived without them. Thank you for making my working and social life enjoyable and for providing intelligent and useful comments of my work.

Very special thanks to my dear friends Lars-Goran Svensk (especially for teaching me how to fish), Fehri Meguedmini and Riadh Baccouche for giving me the occasional illusion that they can create a small Tunisia within Sweden. I also wish to thank my colleagues at the department who made me feel like I belong to a big family: Eva Enefjord, Gunilla Klaar, Torsten Palm, Isabella Scandurra, Maria Hagglund, Anneli Edman, Ingrid Lukell, Torsten Johnson, Anne Hakansson, Ann Gunnarsson, Anna-Lena Kaberg, Vladislav Valkovsky, etc.

I am very grateful to Adela Katchaounova, Gordon Kelly, Thomas and Kajsa, Johan and Anna-Maria, Milena Kopranova and her husband, for their lovely acquaintances.

I would like to thank Julian Richardson for his hospitality, support, and guidance during my stay with him at Harriot-Watt. Similarly, I wish to thank Alan Frisch and Ian Miguel for hosting me, several times, in York and for their support and fruitful collaboration. I wish to express my gratitude to Simon Wrang for doing most of the implementation of the Fiona system as part of his Magister thesis.

I also would like to thank many of my old friends including Imed Hammouda, Mohammed Khames, Nejib Ammar, and wish them the best of luck with their theses.

Last and by no means least, I am very grateful to my parents (to whom this thesis is dedicated), my sister and her husband, my brother, and my relatives (including the 80 plus cousins!) for all the love, support and happiness they bring me.
Declarations

Parts of this thesis are the results of collaboration with Pierre Flener, Alan Frisch, Zeynep Kiziltan, Ian Miguel, Toby Walsh, and Justin Pearson. I declare that I have made a substantial contribution to this work and this thesis is composed by myself. I give more details concerning joint work. In Chapter 4, parts of the work in Section 4.4 has been published in [43], where writing the paper is a result of joint work of all authors, but I proposed the models, after discussing them with Zeynep Kiziltan, and I did all the experimental work by myself. The idea of having function variables has been published in [26] where I am the main contributor and is presented in Chapter 5 in more detail and with new ideas and results. The theoretical work in Section 6.3 in Chapter 6 is a result of a collaboration with Toby Walsh, but I proved all theorems and did all the experimental work. A subset of Chapter 6 has been published in [44]. Pierre Flener has made a great contribution towards writing the grammar of the \( \mathcal{F} \) language. Toby’s valuable feedback on earlier versions significantly improved the thesis. The implementation of the prototype system was mainly performed by Simon Wrang as part of his Magister thesis under my supervision. However, I designed the grammar of both the input and the output language, all the rewrite rules, extended the implementation to cope with multiple function variables and implemented the constraint formulation selection heuristics. Finally some of the ideas in this thesis concerning modelling and symmetry-breaking came from the work on matrix modelling [23] and symmetry in matrix modelling [24], where I made significant contributions.
Contents

1 Introduction ................................................. 11
  1.1 Topic of the thesis ...................................... 11
  1.2 Constraint programming ................................ 12
    1.2.1 Elements of the current constraint programming world . . . 12
    1.2.2 Modelling CSPs as constraint programs ................. 14
    1.2.3 Alternate models .................................... 16
  1.3 Overview of the thesis ................................ 16
  1.4 Contributions of this thesis ......................... 19
  1.5 Structure of the thesis ................................ 20

2 Formal Background ....................................... 21
  2.1 Set-theoretic concepts ................................ 21
    2.1.1 Sets ............................................. 21
    2.1.2 Relations and functions .......................... 22
  2.2 Constraint satisfaction problems ....................... 24
  2.3 Constraint programming ................................ 25
  2.4 Definition of the $L$ language ..................... 26

3 Related Work ............................................. 30
  3.1 The modelling process ................................ 30
    3.1.1 Very similar models ............................ 31
    3.1.2 Quite unsimilar models ......................... 32
  3.2 Modelling languages .................................. 33
  3.3 Summary ............................................. 34

4 Model Abstraction ....................................... 35
  4.1 Modelling idioms ...................................... 35
    4.1.1 Beyond integer and set variables ................. 36
    4.1.2 Definition of the $F$ language ................ 37
  4.2 The graph coloring problem ........................... 39
  4.3 The warehouse location problem ...................... 42
  4.4 The balanced academic curriculum design problem .... 46
    4.4.1 A first model ................................... 48
    4.4.2 A second model .................................. 49
    4.4.3 Combining the models ......................... 50
    4.4.4 Experimental results ............................ 51
    4.4.5 Models based on set variables ................... 53
    4.4.6 A model using function variables ............... 54
  4.5 Summary ............................................. 55
List of Figures

1.1 Some elements of the CP world ................................................. 13
1.2 Function declarations .............................................................. 17
1.3 Function operations and predicate ............................................. 17
1.4 Function properties ................................................................. 18
1.5 Basic elements of the model of the file packing problem using a function variable ......................................................... 18
1.6 A different approach to modelling ............................................. 19
2.1 Relations and functions ............................................................... 22
4.1 Examples of function declarations ............................................. 38
4.2 Function operations and constraints allowed in $\mathcal{F}$ (but not in $\mathcal{L}$) .......................................................... 39
4.3 An $\mathcal{F}$ model of the GCP using a function variable ................ 39
4.4 An $\mathcal{L}$ model of the GCP using a 1d array of integer variables .... 40
4.5 An $\mathcal{L}$ model of the GCP (optimization) using a 1d array of integer variables .................................................. 41
4.6 An $\mathcal{L}$ model of the GCP (optimization) based on a 2d 0/1 array .... 41
4.7 An $\mathcal{F}$ model of the GCP (optimization) using a function variable .... 42
4.8 An $\mathcal{F}$ model of the WLP using a function variable ................ 43
4.9 An $\mathcal{L}$ model of the WLP using a 1d array of integer variables .... 44
4.10 An $\mathcal{L}$ model of the WLP using a 2d array of Booleans ................ 44
4.11 An $\mathcal{L}$ model of the modified WLP using a 1d array of integer variables .................................................. 45
4.12 An $\mathcal{L}$ model of the modified WLP using a 2d 0/1 array ............ 46
4.13 An $\mathcal{F}$ model of the modified WLP using a function variable .... 47
4.14 Inputs to the BACDP ................................................................. 48
4.15 The $\text{CUR}1d$ model of the BACDP ........................................... 49
4.16 Model $\text{CUR}2d$ of the BACDP .................................................. 50
4.17 The $\text{CUR}1d + \text{CUR}2d_{\text{CP}}$ combined model ..................... 52
4.18 The $\text{CUR}1d + \text{CUR}2d_{\text{ILP}}$ combined model ..................... 53
4.19 Finding an optimal solution. N/A stands for not available while a “-” means that no results were found within 1 clock hour. .............. 53
4.20 Proving optimality. N/A stands for not available while a “-” means that no results were found within 1 clock hour. .............. 54
4.21 A model of the BACDP based on set variables ............................ 55
4.22 An $\mathcal{F}$ model of the BACDP .................................................. 56
5.1 Alternative $\mathcal{L}$ representations of a total function variable ......... 58
5.2 Alternative representations of a partial function variable ............... 60
5.3 Alternative representations of an injective total function variable .... 61
5.4 Alternative representations of an injective partial function variable ... 61
5.5 Alternative representations of a surjective total function variable .... 62
7.17 A generated $\mathcal{L}$ model of the WLP using 2d 0/1 array .......................... 133
7.18 A hybrid $\mathcal{L}$ model of the WLP ............................................................... 134
7.19 Generated $\mathcal{L}$ model of the bus driver scheduling problem using 1d array .... 135
7.20 Generated $\mathcal{L}$ model of the bus driver scheduling problem using a 2d 0/1 array ................................................................. 136
7.21 Generated $\mathcal{L}$ model of the bus driver scheduling problem using a 1d array and a 2d 0/1 array ................................................................. 137
7.22 A generated $\mathcal{L}$ model of the L($n,m$) problem ........................................ 138
7.23 A generated $\mathcal{L}$ model of the L($n,m$) problem ........................................ 139
7.24 A generated $\mathcal{L}$ model of the L($n,m$) problem ........................................ 139
7.25 A generated $\mathcal{L}$ model of the L($n,m$) problem ........................................ 140
7.26 A generated $\mathcal{L}$ model of the L($n,m$) problem ........................................ 140
7.27 A generated $\mathcal{L}$ model of the golomb ruler problem .................................... 141
7.28 A generated $\mathcal{L}$ model of the golomb ruler problem .................................... 141
7.29 A generated $\mathcal{L}$ model of the golomb ruler problem .................................... 142
7.30 A generated $\mathcal{L}$ model of the golomb ruler problem .................................... 142
List of Tables

7.1 Keywords .................................................................................................................... 106
7.2 Operator Precedences .................................................................................................. 109
7.3 No. of backtracks (fails) and running time (in milliseconds) to find all solutions to four instances of Langford’s problem. A dash means that no results were collected after 1 hour. ................................................................. 135
Chapter 1

Introduction

The topic of the thesis is introduced in Section 1.1. Constraint programming is briefly presented in Section 1.2 and models of a file packing problem are discussed. We give an overview of the thesis in Section 1.3 and describe our contributions in Section 1.4. Finally, in Section 1.5, we explain the structure of this thesis.

1.1 Topic of the thesis

One of the most important and most studied areas in Artificial Intelligence is the art of modelling. Modelling is the mental process of mapping an informal description of a problem into a formal description in a particular formal system. The formal description will then be processed according to the inference methods provided by the formal system. One of the major challenges is to make the elements of the formal system as close as possible to the elements of the informal one without jeopardizing the efficiency of the formal system. Reducing the conceptual gap between the informal description of the problem and the formal one makes the task of modelling easier.

Many interesting problems arising from real-life are constraint satisfaction problems (CSPs). Examples are production planning subject to demand and resource availability so that profit is maximized, air traffic control subject to safety protocols so that flight times are minimized, transportation scheduling subject to initial and final location of the goods and the transportation vehicles so that delivery time and fuel expenses are minimized, etc. A CSP consists of a set of variables, each with a finite domain of values, and a set of constraints. Each constraint is a relation defining the allowed values for a given subset of variables. A solution to a CSP is an assignment of values to the variables that is consistent with all constraints. Typically, we are interested in whether a solution exists, in finding one or all solutions, or in finding an optimal solution relative to a given cost function.

Constraint programming (CP) is a two-level architecture that allows the study of computational systems based on constraints. At the first level (the constraint part), the set of constraints of the problem is declaratively specified, while describing how to search for a solution is the concern of the second level (the search part). CP languages employ complex constraint-solving algorithms (or: propagation algorithms) that are employed by specifying a set of constraints, and provide non-deterministic constructs to describe the search. Since constraints arise naturally in CSPs, many of these problems can be declaratively expressed as the constraint part of constraint programs. In fact, many CSPs have been effectively modelled as constraint programs and efficiently solved using CP techniques. This is due to CP being declarative and providing high-level abstractions,
which helps reducing the development time of applications by orders of magnitude.

However, the effective modelling of CSPs as constraint programs is both difficult, even for application domain experts, and time-consuming. For many applications, the conceptual gap between the informal problem description and the constraint program is still large. Moreover, many of these problems are ill-behaved, in the sense that it can be shown that solving them requires an amount of time that is worse than polynomial in the size of the input data, hence making solving times prohibitively long. In other words, solving these problems is in general computationally intractable (NP-hard) [51]. Furthermore, effective modelling often requires trying alternate models and selecting a model that efficiently solves the problem, especially that there are many different ways of modelling a CSP as a constraint program. Finally, there is a lack of support for modelling by current CP environments, which makes it difficult for CP to be accessible to a wider range of users than researchers working on this field and domain experts from industry.

This thesis addresses the lack of support for modelling CSPs as constraint programs. In particular, the thesis is focused on the tasks of alternate model generation and model selection, as well as on proposing an architecture for a high-level modelling tool that would assist the modelers with these two tasks.

1.2 Constraint programming

CP has been successfully employed to solve many real-life CSPs. However, effectively modelling a CSP using CP requires a lot of expertise. The modeler’s initial task is to map the elements of the problem domain into some elements of the CP world, the latter being composed of constructs that allow the declaration of variables and constraints on these variables. For example, if the problem formulation in informal language talks about which teacher should teach which course, then a natural mapping is to introduce a variable for each course that takes a value from the set of teachers. After successfully performing the first task, i.e., mapping the elements of the problem domain into some elements of the CP world, the modeler may extend the initial model by describing the search order, or by adding further constraints, or by changing the search and/or the propagation algorithms.

Be it because of a requirements change in the original problem formulation or for efficiency reasons, the modeler may find herself re-doing the initial step and exploring another mapping between the elements of the problem domain and the elements of the CP world. This iterative modelling process goes on till the modeler is satisfied with the outcome.

1.2.1 Elements of the current constraint programming world

Constraint programs are composed of 4 parts, namely the declaration of the instance data (or: inputs), the declaration of the decision variables, the statement of the cost function (or: objective function), if any, and the statement of the problem constraints. In this thesis, we restrict ourselves to finite domains. Furthermore, we only consider domains where the elements are either integers or integer sets. This allows a simpler presentation and the results can be extended to cope with domains of elements of a different type by mapping every element of a different type to an integer.

Any input can be a given integer, a given integer set, a given set of integer sets, a given function from a given integer set into a given integer set, or a given relation between two given integer sets. Two types of decision variables are supported, namely integer variables...
Figure 1.1: Some elements of the CP world

An integer variable takes a value from a given integer set or a given integer interval, while a set variable takes a value from a given set of integer sets. When the domain set is composed of the two elements 0 and 1, such variables are referred to as Boolean variables. The objective function and the constraints on the problem variables and inputs can be stated using arithmetic, logic, and set operators. A set of useful recurring constraints can often be stated more compactly by the use of global constraints [64, 63, 55]. As an example, given n integer variables, the alldifferent global constraint [64] enforces that all variables take distinct values. Some of the elements of the CP world are depicted in Figure 1.1, which shows examples of a Boolean variable, an integer variable, a set variable, some global constraints, as well as some operators, relations, and constraints of arithmetic, logic, and set theory.

Further classes of constraints exist in the literature. An implied constraint is a logical consequence of the initial problem constraints [32]. Through the addition of implied constraints, the amount of search may be reduced due to extra propagation without changing the set of solutions. A redundant constraint [72] on the other hand does give no extra propagation, and a redundant value [4] is any value that does not participate in any solution. A symmetry-breaking constraint [15] is not a logical consequence of the initial problem constraints, but breaks symmetry that arises because either some variables or some values are indistinguishable. The addition of symmetry-breaking constraints reduces the set of solutions by eliminating symmetric ones, which may help improve the search by avoiding symmetric branches. When a problem involves two sets of variables, it may be the case that these sets of variables have to be linked through linking constraints. Finally, when redundant variables are introduced to a model, these variables have to be connected to the original variables of the problem through channelling constraints.
1.2.2 Modelling CSPs as constraint programs

Modelling a CSP as a constraint program requires finding a representation of the problem in terms of decision variables and constraints on these variables. Quite often more than one representation needs to be tried before one can easily and efficiently express all the problem constraints. We illustrate these ideas through an example. Let us try to model a file packing problem as a constraint program. Suppose one wants to copy a set of files from a hard-disk onto as few as possible diskettes of given capacities. The objective is to figure out the minimal number of needed diskettes, and which files are copied to which diskette.

A first model. Assume that the files are given as the set \( F = \{1, \ldots, n\} \) and the available diskettes as the set \( D = \{1, \ldots, m\} \). Then, one way to model the decision variables is as follows. We introduce an integer variable \( F_i \) for each file \( i \) whose domain is the set of diskettes \( D \). The assignment of value \( j \) to variable \( F_i \) means that file \( i \) should be copied to diskette \( j \). One can state the constraint that each diskette has a capacity that should not be exceeded as follows:

\[
\forall j \in D \cdot \sum_{i \in F : F_i = j} size(i) \leq capacity(j)
\]

However, this is not possible in current CP languages because we cannot have a decision variable participating in a condition that appears in a sum expression \((F_i = j)\). One way to overcome this difficulty is to introduce \( n \cdot m \) Boolean variables and add the following channelling constraints:

\[
\forall i \in F \cdot \forall j \in D \cdot B_{i,j} = 1 \iff F_i = j
\]

in which case the Boolean variable \( B_{i,j} \) is bound to 1 if and only if the constraint \( F_i = j \) is satisfied. Now, one can state the capacity constraint with the help of weighted sums as follows:

\[
\forall j \in D \cdot \sum_{i \in F} B_{i,j} \cdot size(i) \leq capacity(j)
\]

To state the objective function we need to have access to the set of actually used diskettes. However, this set is only implicit in our variable modelling. To cure this problem, more variables could be introduced to explicitly represent the set of actually used diskettes. These extra variables should then be connected to the initial ones via linking constraints. For instance, we can introduce \( m \) Boolean variables \( C_j \) to the first formulation. The meaning of \( C_j = 1 \) is that at least one file is copied to diskette \( j \). Now, the objective function minimizes the sum of these Booleans:

\[
\text{minimize } \sum_{j \in D} C_j
\]

However, we need to link these extra Booleans to the initial variables. This can be done through the following linking constraints:

\[
\forall i \in F \cdot \forall j \in D \cdot F_i = j \rightarrow C_j = 1
\]

which make sure that whenever a file is assigned to a diskette, then that diskette is marked as used.
A second model. In the previous model, we needed to introduce more variables and more constraints, so let us try to think of a different way of representing the problem. An alternative way is to introduce \( n \times m \) Boolean variables \( P_{i,j} \) instead. The meaning of \( P_{i,j} = 1 \), for \( i \in F \) and \( j \in D \), is that file \( i \) is copied to diskette \( j \). However, this variable modelling introduces the burden of adding extra constraints:

\[
\forall i \in F : \sum_{j \in D} P_{i,j} = 1
\]

to enforce that each file is copied to exactly 1 diskette. Nevertheless, we can now state the capacity constraint as weighted-sum constraints:

\[
\forall j \in D : \sum_{i \in F} P_{i,j} \times \text{size}(i) \leq \text{capacity}(j)
\]

However, we still find it difficult to state the objective function because the set of used diskettes is not explicit. We can cure this problem in similar ways to the previous model. We introduce the same \( m \) Boolean variables \( C_j \), but state the following linking constraints instead:

\[
\forall i \in F : \forall j \in D : P_{i,j} = 1 \rightarrow C_j = 1
\]

However, this model also introduces more variables.

A third model. Since the second model also introduces more variables, let us again try to think of a new way of representing the problem. We can introduce \( m \) set variables \( S_j \) such that for each \( j \) in \( D \), we have that \( S_j \subseteq F \). For all \( i \in F \), and \( j \in D \), the meaning of \( i \in S_j \) is that file \( i \) is to be copied to diskette \( j \). Again, we need to make sure that each file is to be copied to exactly 1 diskette. This is achieved by making sure that the union of all set variables is the set \( F \), and that they are pairwisely disjoint:

\[
\bigcup_{j \in D} S_j = F
\]

\[
\forall i \in D : \forall j \in D : i \neq j \rightarrow S_i \cap S_j = \emptyset
\]

The capacity constraint can be expressed with help of global weighted-cardinality constraints [39], which enforce that:

\[
\forall j \in D : \sum_{i \in S_j} \text{size}(i) \leq \text{capacity}(j)
\]

The objective function can be expressed by minimizing the number of set variables that are not empty sets as follows:

\[
\text{minimize} \sum_{j \in D : S_j \neq \emptyset} 1
\]

which employs a set variable in the condition in the sum expression \( (S_j \neq \emptyset) \). This expression is an allowed expression in the objective function due to Gervet [39].
1.2.3 Alternate models

Quite often alternate models of a given CSP need to be tried. Different models have different properties and expose different aspects of the problem. We divide the space of alternate models into three classes.

The first class represents the set of alternate models that are quite similar in the sense that an alternate model can be achieved by a simple transformation of an initial model. For instance quite similar alternate models can be achieved by either adding implied constraints and/or symmetry-breaking constraints to an initial model, or restating a problem constraint differently, or removing redundant constraints and/or redundant values from an initial model.

The second class represents alternate models that are quite different in the sense that they employ variables of a different type and have different constraint formulations. For instance, the first formulation of the file packing problem, which uses integer variables, is quite different from the formulation that uses set variables.

Finally, some alternate models are the result of combining two different models of the same problem. This may be necessary in cases where some of the constraints are better specified on one model, while others are better specified on the other model. The combined model has the best constraint formulation from each model, and connects the sets of variables from each model through channelling constraints. For instance, a well studied class of problems are permutation problems [36, 12, 66, 67, 81]. In a permutation problem, all variables have the same domain, the number of variables is equal to the number of values, and each variable must be assigned to a different value. A dual model for permutation problems can be achieved by exchanging the roles of variables and values in the initial model. The primal and dual models can be combined and connected through channelling constraints.

1.3 Overview of the thesis

This thesis addresses the issue of assisting a modeler with the task of modelling CSPs as constraint programs. It helps modelers explore alternate models.

To achieve this, we focus on the class of problems that can be expressed as function problems. A function problem is a CSP where — at the problem description level of abstraction — the objective is to find at least one function from a given set into another given set such that some constraints are satisfied. We introduce proper constructs to allow the statement of function problems. These constructs allow the declaration of function variables. A function variable is a decision variable that takes a value from the set of all possible functions from a given set (source set) into another given set (target set). We also allow function operations to be used to state the problem. Figure 1.2 shows how to declare a function variable. The declaration $F : V \mapsto W$ indicates that the function variable $F$ takes a value from the set of all possible partial functions from source set $V$ into target set $W$, while the declaration $F : V \rightarrow W$ indicates that the function variable $F$ takes a value from the set of all possible total functions from source set $V$ into target set $W$. For instance, when $V = \{1, 2\}$ and $W = \{1, 2\}$, the domain of function variable $F$ that is declared to be partial is:

$$\{\{\}, \{(1, 1)\}, \{(1, 2)\}, \{(2, 1)\}, \{(2, 2)\}, \{(1, 1), (1, 2)\}, \{(1, 2), (2, 2)\},$$

$$\{(1, 2), (2, 1)\}, \{(1, 1), (2, 2)\}\}$$
$F : V \mapsto W$  
$F : V \rightarrow W$  

$F(i)$ function application: the image of $i$ under $F$

$(i,j) \in F$ membership

$\text{domain}(F)$ the set of elements of $V$ that have an image under $F$

$\text{range}(F)$ the set of elements of $W$ returned by $F$ over its domain

$F^{-1}(j)$ the set of elements that have image $j$ under $F$

Figure 1.2: Function declarations

while the domain of $F$ when $F$ is declared to be total is:

$\{(1,1), (2,1), (1,2), (2,2), (1,1), (2,1), (1,2), (2,2)\}$

Figure 1.3 presents operations that are allowed on functions. Assuming $F$ is a function from source set $V$ into target set $W$, the expression $F(i)$ retrieves the image of $i$ under $F$, the predicate $(i,j) \in F$ is true iff pair $(i,j)$ is an element of $F$, the set of elements from the source set $V$ that have an image under $F$ is denoted by $\text{domain}(F)$, the set of elements of target set $W$ returned by $F$ over its domain is denoted by $\text{range}(F)$, and the inverse image set of an element $j$ of target set $W$ is denoted by $F^{-1}(j)$. The reader should not be confused with the domain of a function, denoted by $\text{domain}(F)$, and the domain of a decision variable, which is the set of all possible values for that variable.

Figure 1.4 introduces some properties that hold for some functions. Each property is a constraint that restricts further the set of all possible values of a function variable to the one where all its elements satisfy this extra property. A surjective function $F$ from $V$ into $W$ is a function whose range is $W$. An injective function $F$ from $V$ into $W$ assigns different images for every two distinct elements of its domain. A bijective function is both surjective and injective.

We refer to models expressed using function variables as function models. A function model is at a higher level of abstraction than current CP models, and there is a closer correspondence between the elements of the function model and the elements of the problem description of a function problem. This allows the modelers to model their problems at a suitable level of abstraction. For instance, a possible function model of the file packing problem is as follows. Given the set of files $F$ and the set of diskettes $D$, the objective is to find a total function from $F$ into $D$. Thus, we can declare a function variable $\text{Packing} : F \rightarrow D$ with the meaning that file $i$ is copied into diskette $j$ iff $(i,j) \in \text{Packing}$. To indicate that each diskette has a capacity that should not be exceeded, one might use the following:

$$\forall j \in D \cdot \sum_{i \in \text{Packing}^{-1}(j)} \text{size}(i) \leq \text{capacity}(j)$$

which forces that for each $j$ in $D$, the sum of the sizes of all the elements of $F$ that have $j$ as an image (that is $\text{Packing}^{-1}(j)$) does not exceed the capacity of diskette $j$. The objective function can be expressed as a minimization of the cardinality of the range of
The function $F$ is surjective

The function $F$ is injective

The function $F$ is both injective and surjective

Figure 1.4: Function properties

Figure 1.5: Basic elements of the model of the file packing problem using a function variable

the Packing function:

\[
\text{minimize } |\text{range}(\text{Packing})|
\]

A pictorial representation of the basic elements of the model of the file packing problem using the function variable is shown in Figure 1.5.

Starting with a function model, we propose ways of automatically generating alternate CP models. The automatic generation exploits different representations of a function variable in terms of integer and set variables, employs different constraint formulations, adds implied constraints, and combines different representations in certain ways, which results in combined models. For instance, the three different CP models of the file packing problem proposed in Section 1.2 can be generated from the function model expressed in terms of a function variable. In Figure 1.6, we depict our approach to modelling. Normally, modelers map their problem descriptions into CP models. Instead, we introduce an intermediate abstract layer, which reduces the gap between problem descriptions and models, allowing an easier statement of problems. Also, we propose tool support for automatically producing CP models from a function model.

The translation of a function model may result in more than one CP model. The next natural step is to propose methods that compare these alternate models. We employ a theoretical approach — using the constraint measure proposed by Walsh [81] — that compares different models with respect to the amount of pruning achieved using different consistency techniques as well as different search algorithms. We apply the theoretical approach to models of injection problems, i.e., function problems where the function to be found must be injective. We enhance the theoretical results with an asymptotic analysis.
of the time required to enforce the different consistency levels on the different models and empirically study three different injection problems.

Finally, we propose a modelling tool that takes as input a function model and returns a set of alternate CP models.

1.4 Contributions of this thesis

The main result of this thesis is a methodology for modelling function problems as constraint programs. In order to achieve this result, the following contributions have been established:

- An abstract layer composed of function variables and function operations and constraints is introduced to model function problems at a suitable level of abstraction.

- Having this abstract layer, the automatic generation of alternate CP models becomes possible, which results in the encapsulation of many modelling decisions:
  - The choice of variables.
  - The constraint formulation.
  - The introduction of appropriate extra and dual variables.
  - The introduction of channelling and linking constraints.
  - The integration of different models.
  - The development of heuristics that choose between different constraint formulations.
  - The addition of implied constraints.
With many alternate CP models at hand, we propose some contributions to address the question of model selection:

- An analysis of the task of model selection.
- A theoretical comparison of different models of injection problems supported by an asymptotic analysis and an empirical study.

We put all these results together and build a practical modelling tool. We also evaluate the proposed tool.

### 1.5 Structure of the thesis

The rest of the thesis is structured as follows:

- **Chapter 2** introduces the necessary background knowledge. It explains some set-theoretic concepts, presents CSPs, gives an overview of constraint programming, and introduces a language $\mathcal{L}$ that we use to describe constraint programs throughout this thesis.

- **Chapter 3** reviews related works that address issues related to modelling CSPs as constraint programs.

- **Chapter 4** motivates the need for function variables and their operations. We present our ideas through three CSPs. For each problem we propose different CP models and contrast them with a more natural formulation in terms of function variables. We show that with function variables, the models are easier to understand and modify.

- **Chapter 5** presents the building blocks for the automatic generation of alternate CP models from function models. It also proposes some heuristics for combining models.

- **Chapter 6** is concerned with model selection. We theoretically compare different models of injection problems, and enhance the study with an asymptotic and empirical study. As a result a number of alternate models for injection problems are eliminated.

- **Chapter 7** proposes and evaluates a practical modelling tool. The input to such a tool is a function model expressed at a high level of abstraction. Such a tool proposes alternate CP models by incorporating the results in Chapter 5. Such a tool also uses the contributions presented in Chapter 6 to produce only a selected set of injection models.

- **Chapter 8** concludes and discusses future work.
Chapter 2

Formal Background

In this chapter, we present the necessary background knowledge that is used throughout the thesis. We first present some set-theoretic concepts in Section 2.1. Then, we introduce constraint satisfaction problems in Section 2.2. Finally we provide an overview of constraint programming in Section 2.3 and define a language which we call \( L \) that we will use to describe constraint programs throughout this thesis.

2.1 Set-theoretic concepts

The concepts of sets, functions, and relations are introduced.

2.1.1 Sets

A set is a well defined collection of objects, and will be denoted in this section by a capital letter. The objects comprising the set are called its elements or members. The statement “\( p \) is an element of \( A \)” or, equivalently, “\( p \) belongs to \( A \)” is written

\[ p \in A \]

The negation of \( p \in A \) is also written \( p \notin A \).

We can specify a particular set in two ways. One way is to list its members, if possible. For example,

\[ A = \{1, 2, 3, 4\} \]

denotes the set \( A \) whose elements are the integers 1, 2, 3, and 4. Note that the elements are separated by commas and enclosed in curly braces \{ \}. Alternatively, one can state the properties that characterize the elements in the set. For example,

\[ B = \{x : x \text{ is an integer}, 1 < x\} \]

which reads “\( B \) is the set of \( x \) such that \( x \) is an integer and \( x \) is greater than 1.” denotes the set \( B \) whose elements are the positive integers greater than 1. A letter, usually \( x \), is used to denote an arbitrary member of the set; the colon is read as “such that” and the comma as “and”. The set that has no elements is denoted by \( \emptyset \).

Sets can be finite or infinite. A set is finite if it consists of \( n \) different elements, where \( n \) is some natural number; otherwise a set is infinite. In what follows, we will consider only finite sets.
relations

partial functions

total functions

injections

bijective functions

surjections

figure 2.1: relations and functions

2.1.2 relations and functions

A set \( A \) is a subset of a set \( B \) or, equivalently, \( B \) is a superset of \( A \), written

\[ A \subseteq B \]

iff each element in \( A \) also belongs to \( B \), i.e. if \( x \in A \) implies \( x \in B \). The negation of \( A \subseteq B \) is also written \( A \nsubseteq B \) and thus states that there is an \( x \in A \) such that \( x \notin B \).

Two sets \( A \) and \( B \) are equal, written \( A = B \), iff they consist of the same elements, i.e. if each member of \( A \) belongs to \( B \) and each member of \( B \) belongs to \( A \). The negation of \( A = B \) is also written \( A \neq B \).

The cardinality of a set \( A \), denoted by \(|A|\), is the number of elements belonging to \( A \).

The union of two sets \( A \) and \( B \), denoted by \( A \cup B \), is the set of all elements belonging to \( A \) or \( B \), i.e.,

\[ A \cup B = \{x : x \in A \text{ or } x \in B\} \]

The intersection of two sets \( A \) and \( B \), denoted by \( A \cap B \), is the set of elements which belong to both \( A \) and \( B \), i.e.,

\[ A \cap B = \{x : x \in A, x \in B\} \]

If \( A \cap B = \emptyset \), that is, if \( A \) and \( B \) do not have any elements in common, then \( A \) and \( B \) are said to be disjoint.

The difference of set \( A \) and set \( B \), denoted by \( A - B \), is the set of elements belonging to \( A \) but not to \( B \), i.e.,

\[ A - B = \{x : x \in A, x \notin B\} \]

The cross product of set \( A \) and set \( B \), denoted by \( A \times B \), is the set of all ordered pairs \( \langle a, b \rangle \) where \( a \in A \) and \( b \in B \), in other words,

\[ A \times B = \{\langle a, b \rangle : a \in A, b \in B\} \]

2.1.2 relations and functions

A binary relation \( R \) from a source set \( A \) to a target set \( B \) is a subset of \( A \times B \). For each pair \( \langle a, b \rangle \) in \( A \times B \), either "\( a \) is related to \( b \) via \( R \)" , written \( a \mathrel{R} b \), or "\( a \) is not related to \( b \) via \( R \)".
The domain of a relation $R$ from source set $A$ to target set $B$ is the set of first coordinates of the pairs in $R$ and its range is the set of second coordinates, i.e.,

$$\text{domain}(R) = \{a : \exists b \cdot (a, b) \in R\}$$

$$\text{range}(R) = \{b : \exists a \cdot (a, b) \in R\}$$

The inverse of $R$, denoted by $R^{-1}$, is the relation from $B$ to $A$ defined by

$$R^{-1} = \{(b, a) : (a, b) \in R\}$$

If to each element of the domain set of relation $R$ there is assigned at most one element of the target set $B$, then the collection, say $f$, of such assignments is called a partial function from source set $A$ into target set $B$ and is written

$$f : A \rightarrow B$$

When to each element of the domain set of a relation $R$ there is assigned exactly one element of the target set $B$, then the collection, say $f$, of such assignments is called a total function from source set $A$ into target set $B$ and is written

$$f : A \longrightarrow B$$

The unique element in $B$ assigned to some $a \in A$ by $f$ is denoted by $f(a)$, and called the image of $a$ under $f$. The set of all elements in $\text{domain}(f)$ that have the same image $b$ in $\text{range}(f)$ is denoted by

$$f^{-1}(b) = \{a : a \in A, f(a) = b\}$$

A (partial or total) function $f$ from source set $A$ into target set $B$ is said to be injective if distinct elements in its domain have distinct images, i.e. if

$$\forall a, b \in \text{domain}(f) \cdot a \neq b \rightarrow f(a) \neq f(b)$$

A (partial or total) function $f$ from source set $A$ into target set $B$ is said to be surjective if every $b \in B$ is the image of some $a \in \text{domain}(f)$, i.e. if

$$\forall b \in B \cdot \exists a \in \text{domain}(f) \cdot f(a) = b$$

A (partial or total) function $f$ from source set $A$ into target set $B$ is said to be bijective if $f$ is both injective and surjective.

In general, the inverse relation $f^{-1}$ of a function $f$ from source set $A$ into target set $B$ need not be a function. However, if $f$ is bijective or injective, then $f^{-1}$ is a (total or partial) function from $B$ into $A$ and is called the inverse function.

We present in Figure 2.1 the different varieties of relations and functions and their relationships.

---

1 The long arrow “→” for describing a function should not be confused with the shorter implication logical connective “→”.
2.2 Constraint satisfaction problems

A constraint satisfaction problem (CSP) consists of a set of decision variables, each with a finite domain\(^2\) of values, and a set of constraints. A constraint is a relation defining the allowed values for a given subset of variables. A solution to a CSP is an assignment of values to variables that is consistent with all constraints. Typically, we are interested in whether a solution exists, in finding one or all solutions, or in finding an optimal solution relative to a given cost function. Performing these tasks is in general computationally intractable (NP-hard) [51]. When all the constraints involve at most two variables, then we have a binary CSP (BCSP).

The solution techniques developed for CSPs can be classified as inference (or: problem reduction) and search. These two techniques interact.

Inference techniques may reduce the problem to another equivalent problem, but smaller in size. The size of a CSP is the product of the domain sizes of all variables. The basic idea is to remove from the domains of the variables the values that will not take part of any solution. Such values are said to be inconsistent. Inconsistent values can be detected by using a number of consistency concepts. Different types of consistency guarantee different properties. For instance, the following types of consistency are defined for BCSPs:

- A BCSP is \((i, j)\)-consistent iff it has non-empty domains and any consistent instantiation of \(i\) variables can be consistently extended to \(j\) additional variables [31].

- A BCSP is arc-consistent (AC) iff it is (1, 1)-consistent.

It should be noted that consistency is neither a necessary nor a sufficient condition for a problem to be solvable.

Other types of consistency have been defined for binary constraints [18]. For non-binary constraints, we have the following types of consistency:

- A CSP is generalized arc-consistent (GAC) iff for any value for a variable in a (non-binary) constraint, there exist consistent values for all the other variables in the constraint [56].

- For ordered domains, a CSP is bounds consistent (BC) iff it has non-empty domains and for the minimum and maximum values for any variable in a (binary or non-binary) constraint, there exists at least one consistent value that satisfy the constraint [78].

[18] and [81] compare different consistency properties:

- A consistency property \(A\) is as strong as consistency property \(B\) (written \(A \Rightarrow B\)) iff in any problem in which \(A\) holds we have that \(B\) holds.

- \(A\) is stronger than \(B\) (written \(A \rightarrow B\)) iff \(A \Rightarrow B\) but not \(B \Rightarrow A\).

- \(A\) is incomparable with \(B\) (written \(A \odot B\)) iff neither \(A \Rightarrow B\) nor \(B \Rightarrow A\).

- \(A\) is equivalent to \(B\) (written \(A \leftrightarrow B\)) iff both \(A \Rightarrow B\) and \(B \Rightarrow A\).

The following results are from [18] and [81]:

\(^2\)The domain of a decision variable should not be confused with the domain of a relation
Consistency techniques can be used to reduce a CSP. However, we may also need to perform search in order to actually solve the problem. In [72], the search algorithms are classified into three categories:

- **General search algorithms** do not incorporate any consistency techniques. These search algorithms systematically explore the whole search space. For instance, the *chronological backtracking* (BT) search algorithm assigns a value to one variable at a time and checks for compatibility against all the instantiated variables so far. If failure is encountered, i.e., the assigned value is inconsistent, then a different value is tried. Once all the values are tried and lead to failure, the algorithm backtracks to the previous variable and another value is tried for that variable. The process goes on till either all variables have assigned values or there is no untried value for the first variable. In the second case the problem is **unsatisfiable**.

- **Lookahead algorithms** enforce a consistency property during search. For example, the *forward checking* algorithm (FC) maintains a restricted form of AC that ensures there exist values for the future variables that are consistent with the value assigned to the current variable. FC has been generalized to non-binary constraints [3]. nFC0 makes every k-ary constraint with k−1 variables instantiated AC. nFC1 applies (one pass of) AC to each constraint or constraint projection involving the current and exactly one future variable. nFC2 applies (one pass of) GAC to each constraint involving the current and at least one future variable. Finally, the *maintaining arc-consistency* algorithm (MAC) maintains AC during search, whilst MGAC maintains GAC during search.

- **Gather-information-while-searching algorithms** record the reasons of failure during search and try to avoid searching similar branches in the search tree that would lead to failure for the same reasons. For instance, the *Backjumping* search algorithm (BJ) is very similar to BT, except that when backtracking takes place, the algorithm identifies the assignments that led to failure and backtracks to the most recent variable that is in conflict with the current one.

Further examples of search algorithms are discussed in [72].

### 2.3 Constraint programming

Many problems can be modelled as CSPs and efficiently solved by applying the techniques developed for CSPs. *Constraint programming* (CP) provides a platform for users that helps them describe their problems as CSPs. These languages provide constructs for declaring the variables, their domains, as well as the constraints between these variables. In most of these languages the solving techniques are hidden in a solver that is parameterized by the variable ordering and the value ordering. The variable ordering specifies the order in which the variables are assigned values while the value ordering specifies the order in which the values are assigned to each variable. The variable and value ordering is also referred to as a **labelling strategy**.
A general framework called constraint logic programming (CLP) has been proposed in [46, 79]. Languages like CLP(\text{R}) [47] and Prolog III [14] are based on CLP, but incorporate consistency algorithms. The language CHIP [20] allows the definition of a domain for each variable and uses consistency techniques to reduce the size of the search space. Nowadays, many more (logic) programming languages incorporate libraries that provide constructs for specifying constraints (e.g. sicstus prolog [9]). Also, some languages like oz [71], eclipse [80], ilog solver [45] allow the declaration of set variables as well as constraints on these variables. The domain of a set variable is represented by its greatest lower bound and least upper bound. This representation of set variables is referred to as the interval representation [39]. The consistency algorithms on constraints involving set variables perform reasoning on the bounds of the set domains: the consistency is checked over the lower and upper bound of the domain of set variables [39].

2.4 Definition of the \( \mathcal{L} \) language

Many CP languages are over finite domains. We now present possible ways of specifying the inputs, the variables, the domains, and the constraints on these variables in CP languages like opl [75], ilog solver, sicstus prolog, and eclipse. We provide a generic notation that we call \( \mathcal{L} \) that we will use to describe constraint programs throughout this thesis. The \( \mathcal{L} \) language represents a significant subset of current CP languages.

We use the following syntactic conventions to give the grammar [75]:

- \( \langle nt \rangle \) denotes a non-terminal symbol \( nt \)
- \( \langle object \rangle \) denotes an optional grammar segment \( object \)
- \( \{ object \} \) denotes zero, one, or several times the grammar segment \( object \)
- \( object^+ \) denotes an expression \( object \{ , \ object \} \)
- \( object^* \) denotes an expression \( object \{ ; \ object \} \)

When a nonterminal symbol, say \( \langle n \rangle \), is defined by several rules, say as \( \langle a \rangle \), \( \langle b \rangle \), or \( \langle c \rangle \), we use the notation:

\[
\langle n \rangle \rightarrow \langle a \rangle \rightarrow \langle b \rangle \rightarrow \langle c \rangle
\]

or the notation:

\[
\langle n \rangle \rightarrow \langle a \rangle \mid \langle b \rangle \mid \langle c \rangle
\]

depending on convenience in context.

An \( \mathcal{L} \) model consists of a set of declarations followed by an instruction:

\[
\langle Model \rangle \rightarrow \{ \langle Declaration \rangle \}
\]

\[
\{ \langle Instruction \rangle \}
\]

Declarations are either input declarations or decision variable declarations:
Inputs can be declared to be of the primitive types (namely integers and integer sets), as well as set or function types (built using the cross-product and function type-constructors). The grammar of input declarations, is as follows:

\[\langle\text{InputDecl}\rangle \rightarrow \langle\text{Id}\rangle : \langle\text{Type}\rangle\]

\[\langle\text{Type}\rangle \rightarrow \text{int} \]
\[\rightarrow \text{set}(\text{int}) \mid \text{set}(\langle\text{Id}\rangle)\]
\[\rightarrow \langle\text{Id}\rangle \times \langle\text{Id}\rangle\]
\[\rightarrow \langle\text{Id}\rangle \rightarrow \langle\text{Id}\rangle\]

These input types are represented differently in different CP languages. For instance, in opl integers, sets, as well as arrays of any number of dimensions can be declared. In sicstus prolog integers and lists are used to declare inputs. However, these different representations are different ways of implementing different types of inputs, which are basically integers, sets, and functions.

Different CP languages allow the declaration of different types of decision variables. For instance, while the sicstus prolog finite domain library allows only the declaration of integer variables, ilog solver and eclipse allow the declaration of both integer and set variables. Also, some languages like opl allow the definition of indexed arrays of decision variables.

In L, problem variables can be (arrays of) integer and set variables. An integer variable can have either an integer set or an integer interval (specified by an integer lower bound and an integer upper bound separated by “..”) as a domain. A set variable can be declared as a subset of a given set or as an element of a set of integer sets. The grammar of problem variable declarations in L is:

\[\langle\text{VarDecl}\rangle \rightarrow \langle\text{Id}\rangle \in \langle\text{Id}\rangle\]
\[\rightarrow \langle\text{Id}\rangle \in \langle\text{Integer}\rangle .. \langle\text{Integer}\rangle\]
\[\rightarrow \langle\text{Id}\rangle \subseteq \langle\text{Id}\rangle\]
\[\rightarrow \langle\text{Id}\rangle \mid \langle\text{Id}\rangle^+ \mid \in \langle\text{Id}\rangle\]
\[\rightarrow \langle\text{Id}\rangle \mid \langle\text{Id}\rangle^+ \in \langle\text{Integer}\rangle .. \langle\text{Integer}\rangle\]
\[\rightarrow \langle\text{Id}\rangle \mid \langle\text{Id}\rangle^+ \subseteq \langle\text{Id}\rangle\]

Expressions on integers and sets are constructed from constants, problem inputs, and problem variables, using aggregate operators of arithmetic (\(\sum\)) and sets (\(\cup\) and \(\cap\)), binary (+, −, and *) and unary (+ and −) operators of arithmetic, the cardinality set operator (\(||\)), the max (resp. min) set operator returning the maximum (resp. minimum) element of a set, and the binary set operators (\(\cap\) and \(\cup\)). The aggregate operators must all have a formal parameter that must be within a given finite bound (a set, or an interval, or a function). The grammar of expressions in L is:
Relations are constructed from expressions using the traditional operators and connectives of arithmetic (\(=, \geq, \leq, >, <, \text{ and } \neq\)), sets (\(\in, \notin, \text{ and } \subseteq\)), and logic (\(\neg, \land, \lor, \rightarrow, \text{ and } \leftrightarrow\)). The grammar of relations in \(L\) is:

\[
\langle \text{Relation} \rangle \rightarrow \langle \text{Expression} \rangle \langle \text{ArithOp} \rangle \langle \text{Expression} \rangle \{ \langle \text{ArithOp} \rangle \langle \text{Expression} \rangle \} \\
\rightarrow \langle \text{Expression} \rangle \langle \text{SetOp} \rangle \langle \text{Expression} \rangle \\
\rightarrow \neg \langle \text{Relation} \rangle \\
\rightarrow \langle \text{Relation} \rangle \langle \text{LogicOp} \rangle \langle \text{Relation} \rangle
\]

Constraints on the problem variables are posted using relations (as above), the traditional quantifier \(\forall\) of logic, and global constraints (discussed next). The grammar of constraints is:

\[
\langle \text{Constraint} \rangle \rightarrow \langle \text{Relation} \rangle \\
\rightarrow \forall \langle \text{Id} \rangle \in \langle \text{Id} \rangle \cdot \langle \text{Constraint} \rangle \\
\rightarrow \langle \text{GlobalConstraint} \rangle
\]

A formal parameter is needed for the quantifier \(\forall\). The formal parameter must be within a given finite bound (a set, an integer interval, or a function).

Efficient consistency algorithms are developed for global constraints. We here present those of them that are allowed in \(L\):

- \(\text{alldifferent}(A)\): expresses that all the elements of the array \(A\) of integer variables must take different values [64].
- \(\text{atmost}(Val, A, C)\): is true if some integer \(N\) is the number of elements in the array of integer variables \(A\) that are equal to the integer \(Val\) and \(N \leq C\), where \(C\) is an integer or an integer variable [62].
• **exactly**(Val, A, C): is true if some integer N is the number of elements in the array of integer variables A that are equal to the integer Val and \( N = C \), where C is an integer or an integer variable [62]. The exactly global constraint is also known as **occurs**.

• **atleast**(Val, A, C): is true if some integer N is the number of elements in the array of integer variables A that are equal to the integer Val and \( N \geq C \), where C is an integer or an integer variable [62].

• **min**(A): returns a variable constrained to be equal to the minimal value among the integer variables of array A.

• **max**(A): returns a variable constrained to be equal to the maximal value among the integer variables of array A.

Furthermore, in languages supporting set variables, such as ECLIPSE and ILOG SOLVER, the membership, cardinality, intersection, and union operators of sets are implemented efficiently using global constraints [39]. Also in [39] a **weighted-cardinality** global constraint on set variables is provided. The weighted-cardinality global constraint \( \text{weighted-cardinality}(S, p, q) \) enforces the following:

\[
\sum_{i \in S} p(i) \leq q(S)
\]

where S is a set variable, p is a function returning the integer weight of each i, and q is a function returning the integer capacity of S.

The instruction posts the constraints of the problem, and states the optional cost function whose value has to be optimized. The grammar of instructions is:

\[
\langle Instruction \rangle \rightarrow \text{solve} \ (\langle Constraint \rangle) ;
\rightarrow \text{minimize} \ (\langle Expression \rangle) \ \text{subject to} \ (\langle Constraint \rangle) ;
\rightarrow \text{maximize} \ (\langle Expression \rangle) \ \text{subject to} \ (\langle Constraint \rangle) ;
\]
Chapter 3

Related Work

During the last decades, many efficient algorithms have been developed for solving CSPs [72]. The literature is full of methods that efficiently solve commonly posted constraints, such as the \textit{alldifferent} global constraint. Many search algorithms have been developed as well. However, less attention was given to the question of how to map the problem description into a CSP. For a given description of a problem, the space of possible models is huge. Some of these models are “good”, while others are “bad”, in the sense that given a solution method, the good models are solved more efficiently than the bad ones. Ideally, a modeler should be able to map his/her problem description into the “best” model with respect to a given solution method. For the process of modelling problems as CSPs to be effective, the main elements that affect modelling must be identified and analyzed. Achieving this will also advance the development of measures that compare different models of the same problem.

In Section 3.1, we identify some of the modelling decisions that are made while mapping a problem description into a model. These decisions give us clues on some elements that might affect the model. We review methods that transform a model into another model. The two models have minor differences. We also review different approaches of generating alternate models. The alternate models have major differences and some of them are the result of combining different models. Finally, before summarizing, we discuss in Section 3.2 the modelling languages that come closest to the ideas of this thesis.

3.1 The modelling process

The modelling process maps an informal problem description into a formal model. This requires relating elements of the problem domain (world) to elements of the model domain (CP). Quite often, for a given informal problem description, a huge number of models can be designed.

The modelling process requires making some of the following modelling decisions:

- \textit{The choice of the decision variables}: What to make the variables and what to make their domains.

- \textit{The constraint formulation}: How to state the constraints.

- \textit{The addition of implied constraints} may reduce the size of the search space and is an optional step.
• *The addition of symmetry-breaking constraints* may avoid searching symmetric branches in the search tree and is also an optional step.

• *The development of a labelling strategy* is optional, but may improve dramatically the solving time.

This whole process can then be repeated, leading to the creation of alternate models. This gives rise to the question of model selection, i.e., which of the models is a “good” model with respect to a particular solution method. However, this is a very challenging task, as illustrated in [5, 4] and further in Chapter 6.

In what follows we will present different approaches in the literature related to alternate model generation and selection. We present the related work based on the following classification of alternate models:

• **Very similar models**: Some of the alternate models have a lot of features in common. Their differences are minor as will be demonstrated in Section 3.1.1.

• **Quite unsimilar models**: Opposite to the previous point, the quite unsimilar models share very few features or are the results of combining different models of the same problem as will be shown in Section 3.1.2.

### 3.1.1 Very similar models

Given an initial model of a CSP, alternate models can be generated through the addition of implied and symmetry-breaking constraints, the removal of redundant values and redundant constraints, as well as the reformulation of some of the problem constraints. These transformation methods produce models that are very similar to the initial one. We here present a summary of some of the work that propose such transformation techniques.

Implied constraints (ICs) are logical consequences of the original problem constraints. In the literature two main approaches exist addressing the generation of ICs. Manual approaches take advantage of the knowledge about the problem being solved. For instance, a set of ICs were generated for a car sequencing problem [20]. Getter *et al.* derive effective ICs for solving an online scheduling problem [40]. Proll and Smith employ ICs to enhance the solving of a template design problem [61]. Frisch *et al.* add ICs to an initial model of a steel mill slab design problem [33]. Van Hentenryck discusses many examples in [75].

Despite the usefulness of ICs, very little work has been carried out in the direction of automating their generation. The first attempt is proposed by Borrett and Tsang [4, 5], who introduce a framework for model formulation and selection. They demonstrate the usefulness of the proposed framework on the automatic generation and evaluation of a certain class of ICs, called *composition constraints*. For every group of three variables, a composition IC can be found, provided two binary constraints between the variables exist. Also, an evaluation heuristic has been proposed to assess the usefulness of adding such composition constraints. The design of the evaluation heuristic is based on an extension to Nadel’s theoretical estimates [57]. The approach has been tested on many problems and promising results have been reported. Taking a different approach, Frisch *et al.* propose generating ICs by using automated theorem proving techniques (such as proof planners) [32]. Initial results achieved by employing different methods for generating different classes of ICs are also reported in [32].

Adding ICs to a model can lead to significant reductions in search. However, in some cases, removing some of the constraints, which are redundant, as well as removing
redundant values from the domains of the variables may also be beneficial. Dechter et al. propose the idea of decomposing the problem into independent sub-problems through the removal of redundant constraints [19]. If a constraint graph of a problem is a tree, then a backtrack-free search is guaranteed [30]. Meiri et al. exploit this fact and propose removing redundant constraints in order to reduce the problem to one whose constraint graph forms a tree [52]. In [4], methods are proposed so that such redundant values and constraints are removed.

When some of the variables or some of the values in a CSP are symmetric, the search space contains many symmetric branches. One way to reduce such symmetries is to add symmetry-breaking constraints to an initial model. Also, when some of the problem constraints are badly expressed in the initial model, leading to little propagation, reformulating such constraints may help developing a better model where all the problem constraints are efficiently expressed. Puget has demonstrated the effectiveness of symmetry removal on the Shur problem [59] (Prob015 in CSPLIB at www.csplib.org) and Borrett reports a gain of one order of magnitude in the running time when symmetry breaking constraints are added to the initial model of the Shur problem [4]. In [35], the authors had to reformulate some of the problem constraints in an academic task assignment problem so as to improve the performance. In [24], a collection of problems is presented where a better model is achieved when either symmetry-breaking constraints are added to an initial model, or a better constraint formulation is used for an ill-expressed constraint.

3.1.2 Quite unsimilar models

There are many ways of producing an alternate model. For instance, the variable aggregation transformation [4] merges a pair of variables into a single variable. When this method is applied to a large problem with many variables, the result may be a problem with considerably less variables. The domains and the constraints are also transformed as a result of such a transformation. This may lead to a quite different formulation from the original one. Novello in his Eclipse program\footnote{Available at www.icparc.ic.ac.uk/eclipse/examples/golf.pl.txt} models the Social Golfers Problem by employing a 2-d matrix of sets. An alternate model for the same problem is presented by us in [23]. The alternate model is a modification of Novello’s model in which each set is replaced by a 0/1 vector representing the characteristic function of the set. Our alternate model in [23] allows the reduction of more symmetry than Stephano’s model. An initial model of the Rack Configuration Problem is presented in [75] and alternate models are presented and experimentally compared by us in [48].

Different models of the same problem may have complementary strengths. Thus it might be beneficial to combine the two models so as to generate a new one. We will review the work done on permutation problems, as well as on other problems classes.

A CSP is a permutation problem when all variables have the same domain, the number of variables is equal to the number of values, and each variable must be assigned a different value. An alternate model for permutation problems can be achieved by exchanging the roles of variables and values in the initial model. The first model is referred to as the primal model while the second one is called the dual model. What is interesting is that the dual model is also a permutation problem. The primal and dual models can be combined into a single model. The variables of the two models are linked by channelling constraints.

The integration of different models of permutation problems has been studied by Cheng et al. [12, 13], Smith [66, 67], and Walsh [81], and a similar idea was previously suggested...
by Geelen [36]. There are many ways of combining the primal and dual models. One either poses an *alldifferent* global constraint among the primal variables (resp. dual variables), or a clique of binary not-equals constraints. Furthermore, it has been established that the channelling constraints alone guarantee that any solution is a permutation of the possible values. Hence, we need not pose any extra constraints on the primal or dual variables. Walsh introduced a measure of constraint tightness parameterized by the level of local consistency being enforced, and used it to compare theoretically different models of permutation problems [81]. Walsh also ran some empirical experiments. Maintaining arc-consistency on the channelling constraints achieves more pruning than making the binary not-equals constraints arc-consistent, but not as much as maintaining generalized arc-consistency on the *alldifferent* global constraint. These results are similar when the different models are compared with respect to bounds consistency. Furthermore, these results extend to search algorithms that maintain a restricted form of (generalized) arc-consistency such as forward-checking and algorithms that maintain (generalized) arc-consistency during search (MAC and MGAC). However, the gap may be exponential in the sense that an exponential number of branches may be the difference between two models. Smith [66] states that the principal benefit of combining primal and dual models is to allow the constraints of the problem to be easily expressed in a form that propagates well. The disadvantage, though, is the increase in the number of variables and the number of constraints. This may result in longer running time despite that a smaller search tree may be explored. Smith concludes [66] by stating that:

> It is likely that in any particular case, whether or not combining models is worthwhile can only be established by experiment.

Going beyond permutation problems, in a number of cases, a model based on integer variables and a model based on set variables can be combined. Examples are the Balanced Academic Curriculum Design problem [43] that will be presented in Chapter 4, the Nurse Rostering example in [12], and the Social Golfers problem [66]. Smith shows that combining two models of the Social Golfers problem can reduce the search space [66]. Another problem where combined models are shown to be beneficial is the Progressive Party Problem (prob013 in CSPLIB) [68].

### 3.2 Modelling languages

We review some of the modelling languages for CSPs that are quite related to our work. In Chapter 2, we mentioned the modelling languages CLP, Prolog III, CHIP, sicstus prolog, oz, opl, eclipse, and ilog solver. But, all these languages do not allow the declaration of function variables.

However, as early as 1976, Laurière introduced a modelling language called ALICE to formally state a problem [50]. The ALICE language is characterized by the use of sets, set operators, cartesian product of sets, vectors, matrices, graphs and paths, constants, and functions. Constraints are stated on these mathematical objects using the classical logical connectives ($\land$, $\lor$, $\rightarrow$, $\leftrightarrow$, and $\neg$), the logic quantifiers ($\exists$ and $\forall$), the set operators ($\in$, $\cap$, $\cup$), and the arithmetic operators ($+$, $-$, $\ast$, $/$, $\geq$, $\leq$, $<$, and $\Sigma$).

Edward Tsang and his Constraint Programming Group at Essex University (UK) had two projects related to ours:

- The adaptive constraint satisfaction project [1, 4, 5] aimed at systematically mapping problems, in a dynamic manner, to algorithms and heuristics so as to achieve an
adaptive constraint satisfaction strategy. Thus helping researchers to bring effective algorithms to applications.

- The computer-aided constraint programming [6, 53] project aimed at building a system that encapsulates the entire process of applying CP technology to problems. And by allowing non-expert users to declaratively state their problems, the users may easily experiment with different problem formulations. The system would then assist them to choose an exiting solver, as well as in understanding the underlying technology.

The language NP-SPEC is a logic-based executable specification language [8, 7], which allows the user to specify problems that belong to the NP complexity class. In NP-SPEC, meta-predicates, called tailoring predicates, are used to restrict the extensions of a given predicate. These meta-predicates are subset, partition, permutation, and intfunc, which can be used to capture certain classes of output. The subset predicate has as extensions all subsets of a given set, the partition predicate has as extensions all $n$ subsets of a given set such that these subsets form a partition, the permutation predicate has as extensions all permutations over a given set, and the intfunc predicate has as extensions all functions from a given set into an integer interval.

In [70], a functional specification language called REFINE is used to specify global search problems for a program synthesizer. The REFINE language augments a functional programming language with three type constructors, namely set, sequence, and map, as well as their operations. The set type constructor allows the declaration of variables of set type, the sequence type constructor allows the declaration of variables of sequence type, and the map type constructor allows the declaration of variables of partial function type.

The ALICE, NP-SPEC, and REFINE languages allow variables of types other than integer and integer sets, which allows the statement of problems at a different level of abstraction. One major challenge is how to extend current CP languages to allow variables of new types (not currently supported) such as function variables as well as constraints on these variables.

### 3.3 Summary

Modelling a problem as a CSP in an efficient and effective manner requires a lot of skills. We showed that, in practice, more than one model has to be tried in order to find a good model. In some cases it suffices to add implied constraints to an initial model, in others we need to remove redundant constraints, so as to achieve a good formulation. Some problems have a large amount of symmetry, hence adding symmetry-breaking constraints may enhance the original model. Some constraints can be efficiently and effectively represented by channelling into another model. Alternate models of the same problem may have different strengths, thus it may be beneficial to combine them by linking the variables with channelling constraints. We also reviewed some of the modelling languages and highlighted the features they support.

Except for a very few methods that try to automate some of the tasks in modelling, such as [4, 32], very little work has been done in the direction of helping problem modeler’s with their task. We believe that there is a great need for tools that would assist the modeler in exploring alternate models, as well as in deciding which of these models to select.
Chapter 4

Model Abstraction

In this chapter, our objective is to achieve models that are as close as possible to the informal problem description and beyond the current CP level of abstraction. An upward lift in abstraction leads to a corresponding increase in productivity due to the decrease in programming time and maintenance cost, as well as to making CP accessible to a wider range of users. In particular, we will provide a framework for function problems, where a function from a given set into another given set ought to be found, at the problem description level. We introduce useful recurring modelling idioms that can be captured in new ways. We propose high-level type constructors to capture new types (functions, sequences, and permutations) and identify useful constraints that can be formulated on variables of these types. This framework allows users to express their problem constraints in a uniform way and at a suitable level of abstraction.

We will introduce our ideas through the use of some CSPs, namely a graph coloring problem, a warehouse location problem, and a balanced academic curriculum design problem. We model each problem at two different levels of abstraction. At the current CP level of abstraction, only (arrays of) integer and set variables are supported along with constraints on these variables. We use the language \( L \) (introduced in Section 2.3) to describe the modelling decisions made at the current CP level of abstraction. The alternative level of abstraction we use to reason about problems is introduced with the help of a language that we call \( F \) that is based on set theoretic concepts. The \( F \) language supports function variables and allows the use of function operations and function properties to describe the problem constraints. We refer to models expressed in \( L \) as CP models or simply models, and to models expressed in \( F \) as function models.

For each problem, we compare models written in \( L \) to more abstract formulation in \( F \). We show that, quite often, some useful information is lost in the process of translating an original description of the problem to a model in \( L \), while there exists a close correspondence between the problem statement and the function model expressed in \( F \). We argue that the more abstract \( F \) models are very natural while the \( L \) models are harder to understand and to maintain.

4.1 Modelling idioms

In this section, we present new ways of capturing useful recurring modelling idioms in constraint programs. Then propose proper variables and constraints that describe these idioms.
4.1.1 Beyond integer and set variables

Current CP languages allow decision variables to be (arrays of) either integer variables or set variables. Furthermore, the constraints of the problem are stated as constraints in terms of these variables. So, equipped with (arrays of) integer variables and set variables, what can a problem modeler do? More precisely, are there any useful recurring modelling idioms that can be captured in new ways?

D.R. Smith synthesizes global search (GS) programs from first-order logic specifications [70]. He classifies the GS problems into seven classes based on the type of enumeration involved:

1. Enumerating all subsets of a given finite set.
2. Enumerating all subsets over a given finite integer interval.
3. Enumerating all functions from a given finite set into another given finite set.
4. Enumerating all permutations over a given finite set.
5. Enumerating all sequences of a given (bounded) length \( k \) over a given finite set.
6. Enumerating all sequences over a given finite set.
7. Enumerating all elements of a given finite integer interval (by binary split).

By adapting Smith’s classification and generating instead programs that reflect a constrain-and-generate methodology, one can reuse and exploit the features of constraint solvers [29].

The first and second classes of Smith can be merged into one class when sets are restricted to be of integer type [25]. Since CSPs are over finite domains, the sixth class of Smith is not applicable, while the seventh class is not suitable for CSPs [26]. We thus claim that in many combinatorial optimization problems the objective is to do either of the following [25, 26, 27, 42]:

- **SUBSET**: Find a subset of a given set. For example, finding a clique of a graph amounts to finding a subset of its vertex set, such that every two elements in the subset are connected by an edge.

- **FUNCTION**: Find a function from a given set to another given set. For example, the coloring of the countries of a map, such that any two neighbor countries have different colors, fits this class.

- **PERMUTATION**: Find a permutation of a given set. For example, scheduling jobs according to precedence constraints amounts to finding a permutation of the set of jobs such that the precedence constraints are satisfied.

- **SEQUENCE**: Find a sequence of (bounded) length \( k \) of a given set. For example, a variant of the travelling salesperson problem can be modelled this way, with a set of cities being ordered into a route, such that every city is visited at least once.

The **SUBSET** class can be usefully generalized to \( n \text{SUBSETS} \) [41], where the aim is to find a maximum of \( n \) subsets of the given set. For instance, the coloring problem can also be seen as finding, for each color, the set of countries it colors, i.e., finding \( n \) subsets of
the set of countries, where $n$ is the cardinality of the set of colors. Note that these subsets
must be pairwise disjoint and that their union must be the set of countries; however, this
is not a partitioning problem, as some of the subsets may be empty, denoting the fact
that some colors are not to be used. Hence, the $n\text{SUBSETS}$ class can also be specialized
to $n\text{PARTITION}$, where the aim is to find a partitioning of a given set into $n$ subsets.

CP languages such as CONJUNTO, OZ, and ILOG SOLVER allow the declaration of set
variables to capture problems where a subset of a given set has to be found. We here
investigate making functions, sequences, and permutations first-class concepts.

### 4.1.2 Definition of the $\mathcal{F}$ language

The $FUNCTION$, $PERMUTATION$, and $SEQUENCE$ classes are actually not classes
of stand-alone problems, but rather give rise to powerful high-level type constructors, of
which several can be used in the same program.

Similar to an $L$ model, an $\mathcal{F}$ function model consists of a set of declarations, a set of
constraints, and an optional objective function expression. The declarations of inputs in
$\mathcal{F}$ are the same as the ones for $L$. However, while $L$ supports (arrays of) integer and set
variables, $\mathcal{F}$ supports function variables. The declaration:

$$F : V \mapsto W$$

ensures that $F$ is a function variable that takes a value from the set of all possible partial
functions from $V$ into $W$. The declaration:

$$F : V \rightarrow W$$

declarates a function variable $F$ that takes a value from the set of all possible total functions
from set $V$ into set $W$. The set of values for a partial (resp. total) function variable $F$
can be further restricted to the set of all possible partial injective (resp. total injective)
functions from $V$ into $W$, to the set of all possible partial surjective (resp. total surjective)
functions from $V$ into $W$, or to the set of all possible partial bijective (resp. total bijective)
functions from $V$ into $W$. For instance, in Figure 4.1, we show examples of different function
declarations and pictorially show a possible value for each of such function variables.

The domain$^1$ of a function $F$ is denoted by $\text{domain}(F)$ while the range is denoted by
$\text{range}(F)$. The image of an element $i$ of the domain of $F$ is denoted by $F(i)$ while the
inverse image set of an element $j$ of the range of $F$ is denoted by $F^{-1}(j)$. The relation
$F(i) = j$ is also written $\langle i, j \rangle \in F$. The function operations and constraints allowed in $\mathcal{F}$
are shown in Figure 4.2.

In $\mathcal{F}$, expressions on integers and sets are constructed from constants, problem inputs,
the image of an element of a function variable, the range set of a function variable, the
domain set of a function variable, and the inverse image set of an element of the range of a
function variable, using the aggregate operator of arithmetic $\sum$, the aggregate operators
of sets ($\cap$ and $\cup$), the binary ($+$, $-$, and $*$) and unary ($+$ and $-$) operators of arithmetic,
the cardinality set operator $\|$, returning the number of elements in a set, the $\max$ (resp. $\min$)
set operator returning the maximum (resp. minimum) element of a set, and the
binary set operators ($\cap$ and $\cup$). The aggregate operator $\sum$ must have a formal parameter
that must be within a bound: a given set, or a given integer interval, or a given function,

$^1$The reader should not confuse the domain of a function with the domain of a decision variable.
a function variable, the domain of a function variable, the range of a function variable, the inverse image set of an element of the range of a function variable. The aggregate operators $\cup$ and $\cap$ of sets must have a formal parameter that must be within a given bound (a set or an integer interval).

In $\mathcal{F}$, relations on integers and sets are constructed from expressions using the traditional operators of arithmetic ($=, \geq, \leq, >, <$, and $\neq$), sets ($\in, \notin$, and $\subseteq$), and logic ($\neg, \land, \lor, \rightarrow$, and $\leftrightarrow$).

Finally, constraints on the problem variables are stated in $\mathcal{F}$ using relations (as above), the traditional aggregate quantifier $\forall$ of logic, and constraints that specify certain properties for a function variable $F$ ($\text{injective}(F)$, $\text{surjective}(F)$, and $\text{bijective}(F)$). The aggregate operator $\forall$ must have a formal parameter that must be within a bound: a given set, a given integer interval, a given function, a function variable, the domain of a function variable, the range of a function variable, or the inverse image set of an element of the range of a function variable. The concrete (lower-128 ASCII) grammar for $\mathcal{F}$ is presented in Chapter 7 where appropriate keywords are introduced for the mathematical symbols.

Now we extend $\mathcal{F}$ with type constructors for permutations and sequences. We also present the allowed operations on variables of permutation and sequence types.

**Sequences.** A sequence $S$ of fixed length $k$ over a set $V$ is declared by $S : \text{seq}(V,k)$, whereas a sequence $S$ of bounded length $k$ over a set $V$ is declared by $S : \text{bseq}(V,k)$. The $i^{th}$ element of $S$ is denoted by $S(i)$. The number of elements of $S$ is denoted by $\text{length}(S)$. The set $\text{domain}(S)$ is the set of integers between 1 and $\text{length}(S)$ inclusive, while $\text{range}(S)$ is the same as $\{S(i) : i \in \text{domain}(S)\}$. 
4.2 The graph coloring problem

In the Graph Coloring problem (GCP) the objective is to find a coloring for the vertices of a graph such that any two vertices connected by an edge have different colors.

An F model of the GCP is in Figure 4.3, which is composed of four parts. The set of inputs is specified in the Inputs part, while the set of decision variables in the Outputs part. The objective function, if any, is stated in the Minimize part, otherwise the keyword "None" is used. Finally, the set of constraints is stated in the Constraints part. The semantics of the F model in Figure 4.3 is the following:

$$\forall (\text{vertices}, \text{colors}, \text{edges}) : \text{set(int)} \times \text{set(int)} \times \text{set(\text{vertices} \times \text{vertices})}.$$  
$$\forall \text{COLORING} : \text{vertices} \rightarrow \text{colors}.$$  
$$\text{GCP}((\text{vertices}, \text{colors}, \text{edges}), (\text{COLORING}))$$  
$$(\text{GCP})$$  
$$\forall (a, b) \in \text{edges} \cdot \text{COLORING}(a) \neq \text{COLORING}(b)$$  

Note that we drop the explicit universal quantification symbol on inputs and outputs, and make the objective function explicit. We later also drop the conjunction symbol between the constraints.
The inputs are represented by the integer set \( \text{vertices} \) of vertices, the integer set \( \text{colors} \) of colors, and the set \( \text{edges} \) of edges. The desired output is captured by a total function variable \( \text{COLORING} \) having \( \text{vertices} \) as the source set and the set \( \text{colors} \) as a target set. As the problem is a decision problem, no objective function is specified. However, we restrict any two vertices sharing an edge from having the same image by iterating over all pairs \( \langle a, b \rangle \in \text{edges} \) and disallowing the image of \( a \) under the function \( \text{COLORING} \) to be equal to the image of \( b \) under the same function.

Similar to models in \( \mathcal{F} \), models in \( \mathcal{L} \) are composed of the same 4 parts. For instance, the model shown in Figure 4.4 is a possible formulation of the GCP written in \( \mathcal{L} \). The inputs written in \( \mathcal{L} \) are the same as the ones written in \( \mathcal{F} \). However, \( \mathcal{L} \), as a CP language, supports (indexed arrays of) integer variables and set variables, but not function variables. The output may be represented in \( \mathcal{L} \) by an indexed array of variables ranging over the set of colors. The meaning of \( \text{COLORING}[a] = b \) is that vertex \( a \) gets assigned the color \( b \). The only constraint of the problem is expressed in a similar way as in \( \mathcal{F} \), by iterating over all pairs \( \langle a, b \rangle \) in \( \text{edges} \) and disallowing \( \text{COLORING}[a] \) to take the same value as \( \text{COLORING}[b] \). The model written in \( \mathcal{F} \) and the one written in \( \mathcal{L} \) are to a certain degree similar. However, one major difference is that while \( \text{COLORING} \) is declared to be a total function variable in Figure 4.3, it is declared to be an array of integer variables in Figure 4.4.

Now, let us consider an optimization version of the GCP. Assume that we want to find the coloring that uses the minimum number of colors. Consider the model written in \( \mathcal{L} \) in Figure 4.4. We find it extremely challenging to state the objective function within this model because the set of used colors in the coloring is implicit in the array \( \text{COLORING} \). However, there are at least two ways to overcome this difficulty. The first approach is to introduce more variables to explicitly represent the set of used colors, which will help us state the objective function, and then to link these variables to the variables of the initial model. Therefore, in \( \mathcal{L} \), we have two types of outputs: the desired outputs required by the problem and the auxiliary outputs, which are needed to state some of the problem constraints. The second approach is to change the variable modelling with the hope that an easier formulation of the objective function can be made.

Taking the first approach, one could introduce an auxiliary Boolean array indexed by \( \text{colors} \), say \( \text{USEDCOLORS} \), with \( \text{USEDCOLORS}[b] = 1 \) iff color \( b \) is used by at least one vertex. The objective function can now be stated as a summation on these Boolean variables. However, one has to link the variables in the \( \text{COLORING} \) array with the variables in the \( \text{USEDCOLORS} \) array. This can be achieved by enforcing that whenever a color gets assigned to a vertex \( (\text{COLORING}[a] = b) \) then that color is used \( (\text{USEDCOLORS}[b] = 1) \). Figure 4.5 shows a model reflecting this approach for the optimization version of the GCP.
Taking the second approach, one could model the variables of the GCP by using a 2d 0/1 array \( \text{COLORING} \) indexed by \( \text{vertices} \) and \( \text{colors} \), with \( \text{COLORING}[a, b] = 1 \) iff vertex \( a \) is colored with the color \( b \). However, with this variable modelling, we are still unable to easily state the objective function, unless we again introduce the 0/1 array \( \text{USEDCOLORS} \). In addition, we introduce the burden of reformulating the problem constraints as well as enforcing that each vertex must be assigned exactly one color because the variable modelling fails to guarantee that. The model in Figure 4.6 is a possible formulation of the optimization GCP based on a 2d 0/1 array of variables.

In contrast to the models written in \( \mathcal{L} \), the function model written in \( \mathcal{F} \) (Figure 4.3) can easily be updated to capture the objective function. Since the range of the function \( \text{COLORING} \) denotes all the colors that participate in the function, we simply have to minimize the cardinality of the range of the \( \text{COLORING} \) function. A possible \( \mathcal{F} \) model for the optimization GCP is shown in Figure 4.7.

So far, we have argued that the \( \mathcal{L} \) models of the GCP are harder to update compared to the function model. In the \( \mathcal{L} \) models, we had to introduce more decision variables and

\begin{verbatim}
Inputs: vertices : set(int)
colors : set(int)
edges : set(vertices × vertices)
Outputs: COLORING[vertices] ∈ colors
Auxiliary: USEDCOLORS[colors] ∈ 0..1
Minimize: ∑c∈colors USEDCOLORS[c]
Constraints:
∀(a, b) ∈ edges.
COLORING[a] ≠ COLORING[b]
∀a ∈ vertices · ∀b ∈ colors.
COLORING[a] = b → USEDCOLORS[b] = 1
\end{verbatim}
more constraints (linking constraints) to the model so as to be able to state the objective function. In contrast, in the function model, it is straightforward to state the objective function. Furthermore, the resulting $L$ models for the optimization GCP are harder to understand than the function model. By inspecting the $L$ models, it takes some effort to reconstruct that actually a function from the set of vertices into the set of colors has to be found, and that the Boolean array is introduced to capture the subset of the colors representing the range of the function. On the other hand, the function model is a more natural formulation of the optimization GCP, where the desired function to be found is explicitly declared as a variable. We believe that the function models are at a higher level of abstraction than the $L$ models for the (optimization) GCP as there is a closer correspondence between the problem statements and the function models.

The differences between the $L$ and $F$ models are due to the following differences between $L$ and $F$:

- **Function variables** can be declared in $F$ but not in $L$.
- **Function operations and properties** can be used to concisely state constraints in $F$.
- **Set expressions** are allowed to appear in the objective function of the $F$ models while only expressions on integer variables are allowed in the objective function of $L$ models.

Having function variables and allowing modelers to state their problem constraints using function operations and constraints leads to high-level models that are easy to read and to maintain.

### 4.3 The warehouse location problem

In the Warehouse Location problem (WLP) [75], a company considers opening warehouses on some candidate locations in order to supply its existing stores. Each possible warehouse has the same maintenance cost, and a capacity designating the maximum number of stores that it can supply (Constraint $C_1$). Each store must be supplied by exactly one open warehouse (Constraint $C_2$). The supply cost to a store depends on the warehouse. The objective is to determine which warehouses to open, and which of these warehouses should supply the various stores, such that the sum of the maintenance and supply costs is minimized.

A possible formulation of the WLP in $F$ is shown in Figure 4.8. The inputs are the integer $maintcost$ capturing the maintenance cost, the integer sets of warehouses ($warehouses$) and stores ($stores$), the capacity function storing the capacities of the

![Figure 4.7: An $F$ model of the GCP (optimization) using a function variable](image-url)
 warehouses, and the supplycost function storing the supply cost for each pair of store and warehouse. The only output is the function variable SUPPLIER capturing which store get supplied by which warehouse. A natural formulation of the cost function is stated: We compute the total supply cost by summing the supply cost of each pair in the function SUPPLIER, and add to it the maintenance cost of the warehouses belonging to the range of SUPPLIER. Recall that in \( F \) one can iterate over elements of a function variable as seen in the first part of the expression of the objective function. As for the constraints, Constraint \( C_1 \) is expressed by enforcing that for each warehouse \( j \), the cardinality of the set of stores that have image \( j \) is less than the capacity of \( j \). Recall that SUPPLIER\(^{-1}(j)\) represents the set of all stores that have \( j \) as an image. We completely capture Constraint \( C_2 \) by having SUPPLIER a function from stores into warehouses. Finally, the set of warehouses to open is the range of the SUPPLIER function.

Now, let us try to model the WLP in \( L \). In [75], two OPL models of the WLP are proposed. The first model (Figure 4.9) is based on a 1d array indexed by the stores and ranging over the warehouses. The second model (Figure 4.10) is based on a 2d 0/1 array indexed by the stores and the warehouses. In both models, and in similar ways to the optimization GCP, a 1d 0/1 array OPENW indexed by warehouses is introduced to model the subset of open warehouses and facilitate the formulation of the objective function. The 0/1 array OPENW is linked to the SUPPLIER array in both models, in similar ways as USED_COLORS is linked to COLORING in the optimization GCP.

The capacity constraint is expressed with the help of the global constraint atmost in the model shown in Figure 4.9. However, it is expressed using a row sum expression in the model based on 2d 0/1 array in Figure 4.10.

Our next goal is to check how easy it will be to maintain these different models of the WLP when the problem requirements undergo a minor change. Suppose we modify the WLP as follows (with modifications being highlighted in italics): A company considers opening warehouses on some candidate locations in order to supply its existing stores, as well as possibly closing some of these stores, but maintaining a certain minimum number of stores remains open (Constraint \( C_3 \)). Each possible warehouse has the same maintenance cost, and a capacity designating the maximum number of stores that it can supply (Constraint \( C_1 \)). Each store that is not closed must be supplied by exactly one open warehouse (Constraint \( C_2 \)). The supply cost to a store depends on the warehouse. The objective is to determine which warehouses to open and which stores not to close, and
which of these warehouses should supply the various stores that are not closed, such that the sum of the maintenance and supply costs is minimized.

The \( \mathcal{L} \) model in Figure 4.9 uses integer variable \( \text{SUPPLIER}[i] \) to denote the warehouse that supplies store \( i \). This variable modelling enforces that each store must be supplied by some warehouse. Thus, it is quite difficult to reflect the change in the requirements that some stores, which are to be closed, ought not to be supplied by any warehouse! In order to reflect the changes on the 1d array, we need to introduce a 1d 0/1 array \( \text{OPENS} \) indexed by \( \text{stores} \) and representing the subset of stores to be kept open, i.e., \( \text{OPENS}[i] = 1 \) iff store \( i \) is open. The two arrays together represent our desired output only under the following meaning: the pair \( \langle i, j \rangle \) represents that store \( i \) is supplied by warehouse \( j \) iff \( \text{SUPPLIER}[i] = j \) and \( \text{OPENS}[i] = 1 \). With this meaning, the statement of the problem constraints ought to be changed, resulting in a less efficient formulation thereof because the constraints are of a higher arity. Figure 4.11 depicts a modified version of the \( \mathcal{L} \) model in Figure 4.9. The objective function had to be updated
Inputs:
- maintcost : int
- minnbstores : int
- stores : set(int)
- warehouses : set(int)
- capacity : warehouses ----> int
- supplycost : stores × warehouses ----> int

Outputs:
- SUPPLIER[stores] ∈ warehouses

Auxiliary:
- OPENW[warehouses] ∈ 0..1
- OPENS[stores] ∈ 0..1
- B[warehouses, stores] ∈ 0..1

Minimize:
\[ \sum_{i \in \text{stores}} \text{supplycost}(i, \text{SUPPLIER}(i)) \times \text{OPENS}(i) + \sum_{j \in \text{warehouses}} \text{OPENW}(j) \times \text{maintcost} \]

Constraints:
\[ \forall i \in \text{stores} \cdot \forall j \in \text{warehouses} \\
\text{SUPPLIER}(i) = j \land \text{OPENW}(j) = 1 \rightarrow \text{OPENW}(j) = 1 \]
\[ \forall j \in \text{warehouses} \cdot \forall i \in \text{stores} \\
B[j, i] = 1 \leftrightarrow \text{SUPPLIER}(i) = j \land \text{OPENW}[i] = 1 \]
\[ \forall j \in \text{warehouses} \cdot \sum_{i \in \text{stores}} B[j, i] \leq \text{capacity}(j) \times \text{OPENW}(j) \]
\[ \sum_{i \in \text{stores}} \text{OPENS}(i) \geq \text{minnbstores} \]

Figure 4.11: An L model of the modified WLP using a 1d array of integer variables to reflect that we count only the supply costs of open stores. The linking constraints now state that a warehouse \( j \) is open (\( \text{OPENW}(j) = 1 \)) only when a store \( i \) is both open (\( \text{OPENS}(i) = 1 \)) and supplied by warehouse \( j \) (\( \text{SUPPLIER}(i) = j \)). The capacity constraint can no more be stated using the global constraint \textit{atmost}, instead we have to introduce Boolean variables \( B[j, i] \) that get assigned the value 1 when the constraint \( \text{SUPPLIER}(i) = j \land \text{OPENS}[i] = 1 \) is satisfied. A summation expression on the extra Booleans \( B[w, s] \) is used to state the capacity constraint. Note that we have to re-interpret the Boolean truth values of \( \text{SUPPLIER}(i) = j \land \text{OPENS}[i] = 1 \) as numbers in order to correctly state the capacity constraint. Furthermore, the extra constraints:
\[ B[j, i] = 1 \leftrightarrow \text{SUPPLIER}(i) = j \land \text{OPENW}[i] = 1 \]

are in general delayed because the constraint \( \text{SUPPLIER}(i) = j \land \text{OPENS}[i] = 1 \) is added to the rest of the problem constraints only when the Boolean \( B[j, i] \) is set to 1. Similarly, it is only after \( B[j, i] \) is set to 0 that the negation of \( \text{SUPPLIER}(i) = j \land \text{OPENS}[i] = 1 \) is added to the rest of the problem constraints. This results in delayed propagation, which may lead to poor performance. Restricting the number of open stores is stated using a summation expression on the elements of the array \( \text{OPENW} \).

Modifying the L model of the WLP in Figure 4.10 results in the model in Figure 4.12. The set of open stores is again modelled by a 1d 0/1 array \( \text{OPENW} \) such that \( \text{OPENW}[s] = 1 \) iff store \( s \) is open. Because of the change in the requirements, not all stores are supplied by some warehouse, therefore we need to reformulate the constraint:

\[ \forall i \in \text{stores} \cdot \sum_{j \in \text{warehouses}} \text{SUPPLIER}(i, j) = 1 \]
Inputs:

- maintcost : int
- minnbstores : int
- stores : set(int)
- warehouses : set(int)
- capacity : warehouses -> int
- supplycost : stores x warehouses -> int

Outputs:

- SUPPLIER[stores, warehouses] ∈ 0..1

Auxiliary:

- OPENW[warehouses] ∈ 0..1
- OPENS[stores] ∈ 0..1

Minimize:

\[
\sum_{i \in \text{stores}} \sum_{j \in \text{warehouses}} \text{supplycost}((i, j)) \times \text{SUPPLIER}[i, j] + \sum_{j \in \text{warehouses}} \text{OPENW}[j] \times \text{maintcost}
\]

Constraints:

\[
\forall i \in \text{stores} \\
\sum_{j \in \text{warehouses}} \text{SUPPLIER}[i, j] = \text{OPENS}[i] \\
\sum_{i \in \text{stores}} \text{OPENS}[i] \geq \text{minnbstores} \\
\forall i \in \text{stores} \cdot \forall j \in \text{warehouses} \\
\text{SUPPLIER}[i, j] = 1 \rightarrow \text{OPENW}[j] = 1 \\
\forall j \in \text{warehouses} \\
\sum_{i \in \text{stores}} \text{SUPPLIER}[i, j] \leq \text{capacity}(j) \times \text{OPENW}[j]
\]

Figure 4.12: An L model of the modified WLP using a 2d 0/1 array

into:

\[
\forall i \in \text{stores} \cdot \sum_{j \in \text{warehouses}} \text{SUPPLIER}[i, j] = \text{OPENS}[i]
\]

Restricting the number of stores to be kept open is expressed as a sum constraint.

In contrast to the L models, reflecting the changes to the F model is achieved by declaring that SUPPLIER is a partial function rather than a total one. Hence, we capture by our new variable modelling that actually a total function from a subset of stores into the set of warehouses is to be found. Furthermore, we restrict the cardinality of the domain of SUPPLIER to be greater than or equal to minnbstores.

In the WLP, the minor change in the requirements made it quite difficult to update the L model in Figure 4.9 and forced us to remodel some of the previous constraints in the L model in Figure 4.10. In contrast, the F model is updated in a straightforward manner by declaring SUPPLIER as a partial function. Furthermore, the resulting L models in Figure 4.11 and Figure 4.12 are less intelligible than the function model in Figure 4.13. There is a closer correspondence between the function models of the (modified) WLP and the informal problem statement because F offers commonly encountered abstractions in informal descriptions.

### 4.4 The balanced academic curriculum design problem

We consider the Balanced Academic Curriculum Design Problem (BACDP) proposed in [11] and further studied in [43]. The objective is to design a balanced academic curriculum by assigning courses to periods in such a way that the academic load of each period will be balanced, i.e., as similar as possible. The curriculum must obey the following:
Academic curriculum: an academic curriculum is defined by a set of courses and a set of prerequisite relationships among them.

Number of periods: all the courses must be assigned within a given maximum number of academic periods.

Academic load: each course has associated a number of credits or units that represent the academic effort required to successfully take it.

Prerequisites: courses can have other courses as prerequisites.

Minimum academic load: a minimum amount of academic credits per period is required to consider a student as full time.

Maximum academic load: a maximum amount of academic credits per period is allowed in order to avoid overload.

Minimum number of courses: a minimum number of courses per period is required to consider a student as full time.

Maximum number of courses: a maximum number of courses per period is allowed in order to avoid overload.

The goal is to assign a period to every course in a way that the minimum and maximum academic load for each period, the minimum and maximum number of courses for each period, and the prerequisite relationships are satisfied. An optimal balanced academic curriculum minimizes the maximum academic load of all periods.

Our concerns in this section will be focused on efficiency. We will try to come up with an $L$ model that efficiently solves at least all three real-life instances proposed in [11]. We will show that in order to achieve an efficient model, we will lose in terms of readability of the models. The resulting models are complex and require advanced modelling skills. Finally, we present a function model of the BACDP, which is more natural than all the proposed $L$ models, without regard to efficiency.
A possible \( \mathcal{L} \) model of the BACDP is in Figure 4.15 where the inputs are in Figure 4.14. The inputs are the integer sets \( \text{courses} \) and \( \text{periods} \) giving the courses and periods respectively, the integers \( a \) and \( b \) giving the minimum and maximum allowed academic loads per period respectively, the integers \( c \) and \( d \) giving the minimum and maximum allowed numbers of courses per period respectively, the function \( \text{credit} \) giving the number of credits for each course, and the set \( \text{prereq} \) containing the pairs of courses \( (i, j) \) such that course \( i \) is a prerequisite of course \( j \).

The assignment of periods to courses is modelled using a 1d array \( \text{CUR}1d \) indexed by courses and ranging over periods. The assignment \( \text{CUR}1d[i] = j \) means that course \( i \) is to be given in period \( j \). The academic load for all periods is represented by a 1d array of integer variables \( \text{LOAD} \). Enforcing that each course is to be given in exactly one period is captured by the 1d array \( \text{CUR}1d \). To compute the academic load for each period, one could state

\[
\forall j \in \text{periods} \cdot \text{LOAD}[j] = \sum_{i \in \text{courses}} \text{credit}(i) \quad \text{if} \quad \text{CUR}1d[i] = j
\]

However, this is not an acceptable constraint in \( \mathcal{L} \) because we cannot have a constraint, such as \( \text{CUR}1d[i] = j \), as a condition in a sum expression. In fact, it is not possible to express this constraint in any of the current CP languages (e.g., OPL, ECLIPSE) unless the modeler either explicitly introduces extra Boolean variables and extra constraints such as in ECLIPSE or implicitly such as in OPL. Hence, to compute the load for each period, the Booleans \( B[i, j] \) are introduced along with a set of channelling constraints of the form

\[
B[i, j] = 1 \iff \text{CUR}1d[i] = j
\]

which makes it possible to compute the load for each period with the help of a weighted-sum expression. The integer variable \( C \) is bound to the maximum of the academic load for all periods with the help of the global constraint \( \text{max} \). The objective function minimizes \( C \). The prerequisites constraints are stated by enforcing an ordering on the courses that have a prerequisite relationship. Inequalities are used to restrict the load for each period, while the global constraints \( \text{atleast} \) and \( \text{atmost} \) are used to enforce the restrictions on the amount of courses allowed per period.

One disadvantage of the model in Figure 4.15 is the introduction of \( m \times n \) extra Boolean variables and \( m \times n \) extra constraints in order to be able to compute the academic load for each period. We abuse the notation and refer to the model in Figure 4.15 as \( \text{CUR}1d \).
Outputs:
- \( \text{LOAD}[\text{periods}] \in 0..\text{maxint} \)
- \( \text{CUR1d}[\text{courses}] \in \text{periods} \)

Auxiliary:
- \( C \in 0..\text{maxint} \)
- \( B[\text{courses}, \text{periods}] \in 0..1 \)

Minimize:
- \( C \)

\% the academic load computation
- \( \forall i \in \text{courses} \cdot \forall j \in \text{periods} \)
  - \( B[i, j] = 1 \iff \text{CUR1d}[i] = j \)
- \( \forall j \in \text{periods} \cdot \text{LOAD}[j] = \sum_{i \in \text{courses}} B[i, j] \cdot \text{credit}(i) \)

\% \( C \) is the maximum academic load
- \( C = \max(\text{LOAD}) \)

Constraints:
\% the prerequisites constraints
- \( \forall (i, j) \in \text{prereq} \cdot \text{CUR1d}[i] < \text{CUR1d}[j] \)

\% the academic load constraints
- \( \forall j \in \text{periods} \cdot a \leq \text{LOAD}[j] \leq b \)

\% the course number constraints
- \( \forall j \in \text{periods} \cdot \text{atleast}(j, \text{CUR1d}, c) \)
- \( \forall j \in \text{periods} \cdot \text{atmost}(j, \text{CUR1d}, d) \)

Figure 4.15: The CUR1d model of the BACDP

### 4.4.2 A second model

Another way to model the BACDP in \( L \) is shown in Figure 4.16, where the inputs are shown in Figure 4.14.

The assignment of periods to courses is represented by a 2d 0/1 array of decision variables (\( \text{CUR2d} \)). The meaning of \( \text{CUR2d}[i, j] = 1 \) is that course \( i \) is assigned period \( j \). The academic load for all periods is represented by a 1d array of integer variables (\( \text{LOAD} \)). The maximum academic load of all periods is represented by the integer variable \( C \). The objective function minimizes \( C \). The first constraint enforces that every course is assigned only one period because a 2d 0/1 array on its own does not ensure this property. The second constraint uses a weighted column sum expression to compute the academic load of each period. The third constraint guarantees that \( C \) is the maximum academic load of all periods. Enforcing the prerequisites constraint is however tricky. If course \( i \) is a prerequisite of course \( j \), the fourth constraint implies a strict lexicographical ordering between the \( i \text{th} \) row and the \( j \text{th} \) row of the 2d 0/1 array. Hence, this disallows the course \( j \) to be assigned a period that has lower index than the one assigned to course \( i \). Enforcing the constraints on the academic load and the amount of courses allowed per period is achieved through two sets of inequalities in the fifth and sixth constraints, respectively.

In this model, the academic load constraint of a period is stated by a weighted column sum on the 2d 0/1 array while we had to introduce extra Boolean variables and extra constraints in model \( \text{CUR1d} \). However, while the prerequisites constraints are stated using binary constraints in model \( \text{CUR1d} \), constraints of arity \( 2 \times m \) are used in this model. However, note that the sizes of the domains are bigger in model \( \text{CUR1d} \) than those of the
Boolean variables in this model. Furthermore, the restrictions on the number of courses are expressed using global constraints in model CUR1d while row-sum expressions are employed in this model.

Another important remark about this model is that all the constraints are linear. So we might as well consider solving this model by employing integer linear programming (ILP) techniques [82]. An ILP model can be relaxed into a linear program by dropping the integrality constraints. There exist methods for solving linear programs in polynomial time such as the Simplex method. An ILP model can be solved by relaxing the integrality requirements and solving the relaxed problem using linear programming (LP) techniques [17]. The relaxation provides a lower bound for the optimal solution and provides a point in space where search can start in order to find an optimal integral solution. This solution method is referred to as branch-and-bound search in ILP.

We abuse our notation and refer to the model in Figure 4.16 as CUR2d_d when the model is solved using a CP solver and as CUR2d_dILP when an ILP solver is used instead.

### 4.4.3 Combining the models

The model CUR1d that is based on a 1d array of integer variables and the models CUR2d_dILP and CUR2d_dCP that are based on a 2d array of Boolean variables have complementary strengths.
The models \textit{CUR2dCP} and \textit{CUR2dILP} differ only in the solution methods used. Each method has its own ability to reason with the constraints. Moreover, the academic load constraints for each period are better stated in the models \textit{CUR2dCP} and \textit{CUR2dILP} than in model \textit{CUR1d} because we introduce more Boolean variables and more constraints to state the constraint in the same way (by a weighted-sum expression) in \textit{CUR1d}. The situation is the other way around if we consider the prerequisites constraints. In \textit{CUR1d}, this constraint is stated by ordering the courses that have a prerequisite relationship resulting in binary constraints, while in the model \textit{CUR2dCP} (and \textit{CUR2dILP}) we need to impose partial row sum constraints, which results in constraints of arity $m \times n$. Furthermore, in \textit{CUR1d}, the restrictions on the number of courses for each period are stated using global constraints that maintain generalized arc-consistency during search. In order to benefit from the effectiveness of each model, we propose combining the models \textit{CUR1d} and \textit{CUR2dCP} into a model called \textit{CUR1d+CUR2dCP}, and the models \textit{CUR1d} and \textit{CUR2dILP} into a model called \textit{CUR1d+CUR2dILP} by channelling the variables of the participating models. The disadvantages of these combinations are the increased number of variables, and the presence of additional channelling constraints that are to be processed.

In \textit{CUR1d+CUR2dCP}, we pose the academic load constraints on the 2d array, and all the other constraints on the 1d array. This allows \textit{CUR2dCP} to benefit from \textit{CUR1d}'s global constraints and prerequisites constraints, and \textit{CUR1d} benefits from \textit{CUR2dCP}'s effectiveness of the academic load constraints. The combined model is shown in Figure 4.17.

In \textit{CUR1d+CUR2dILP}, we specify the prerequisites constraints on the 1d array, and all the other constraints on the 2d array. This allows \textit{CUR2dILP} to benefit from \textit{CUR1d}'s power in the statement of the prerequisite constraints, and \textit{CUR1d} benefits from \textit{CUR2dILP}'s power in the statement of the academic load constraints. Moreover, this combination allows a hybrid solution method to be employed to solve the problem. The combined model is shown in Figure 4.18.

\subsection{Experimental results}

It is quite difficult to see which of the proposed $\mathcal{L}$ models of the BACP is best. Also it is difficult to see if there was any need for integrating multiple models as well as employing multiple solution methods (CP and ILP). To test this, we ran some experiments to compare the proposed models \textit{CUR1d}, \textit{CUR2dCP}, \textit{CUR2dILP}, \textit{CUR1d+CUR2dCP}, and \textit{CUR1d+CUR2dILP}. All models are implemented in opl [75]. Figure 4.19 and Figure 4.20 show the results on the three real-life instances used in [11] to find an optimal solution and prove optimality, respectively. In the instances, we have 8, 10, and 12 periods, and 46, 42, and 66 courses, respectively. We use the same labelling strategy for \textit{CUR2dCP} as in [11], which groups the variables by periods and assigns the value 1 first. Note that this branching heuristic achieves the best results in [11] where only the model \textit{CUR2dCP} and the model \textit{CUR2dILP} are compared. As for \textit{CUR1d}, we use the \textit{fail first} labelling strategy, i.e., branching on the variable with the smallest domain, and choosing values in lexicographical order.
The quickest model that found the optimal solution is $CUR1d + CUR2d_{ILP}$, which shows that a hybrid model resulted in a better model despite the increased number of variables and additional channelling constraints. In this hybrid integration, the CP model is essential in reducing the search space while the ILP model with its relaxation is essential for bounding and guiding the search. The second best model is the ILP model $CUR2d_{ILP}$, which outperformed all the CP models. Finally, the integrated CP model $CUR1d + CUR2d_{CP}$ has less runtime than either of the CP models. This is due to the increase in the amount of pruning, which led to a reduction in the search space that compensated the increase of variables and constraints.

Proving optimality was tough for all CP models. We observe that $CUR1d + CUR2d_{ILP}$ proves optimality quicker than $CUR2d_{ILP}$. This is because it benefits from the CP model in reducing the search space, and from the relaxation of the ILP model, which bounds and guides the search.

The CP solver used in the experiments is ILOG Solver, while the ILP solver is the CPLEX one (both used by OPL). In [11], oz was used to solve the model $CUR2d_{CP}$, and lp-solve\(^3\) to solve the model $CUR2d_{ILP}$. Their experiments are concerned with finding an optimal solution but not with proving optimality. They showed that with varying the default labelling heuristic of the model $CUR2d_{CP}$, the three instances were solved very quickly, but lp-solve could only solve the first instance. However, our experiments using OPL showed the opposite, as seen in Figure 4.19. Note that we use the same labelling strategy for the model $CUR2d_{CP}$ as [11].

\(^3\) An ILP solver: available at ftp://ftp.ics.ele.tue.nl/pub/lp_solve
Outputs: $LOAD[periods] \in 0..maxint$
$CUR1d[courses] \in periods$

Auxiliary: $C \in 0..maxint$
$CUR2d[courses, periods] \in 0..1$

Minimize: $C$

% the academic load computation
$\forall j \in periods \cdot LOAD[j] = \sum_{i \in courses} CUR2d[i, j] \cdot credit(i)$

% $C$ is the maximum academic load
$C = \max(LOAD)$

% the prerequisites constraints
$\forall (i, j) \in prereq \cdot CUR1d[i] < CUR1d[j]$

% the academic load constraints
$\forall j \in periods \cdot a \leq LOAD[j] \leq b$

% the course number constraints
$\forall j \in periods \cdot c \leq \sum_{i \in courses} CUR2d[i, j] \leq d$

% channelling constraints between the 1d array and 2d 0/1 array
$\forall i \in courses \cdot \forall j \in periods \cdot CUR1d[i] = j \leftrightarrow CUR2d[i, j] = 1$

Figure 4.18: The $CUR1d + CUR2d_{ILP}$ combined model

<table>
<thead>
<tr>
<th>Model</th>
<th>Results</th>
<th>8 periods</th>
<th>10 periods</th>
<th>12 periods</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CUR1d$</td>
<td>runtime</td>
<td>30.39</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>failures</td>
<td>302755</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$CUR2d_{ILP}$</td>
<td>runtime</td>
<td>1.29</td>
<td>3.45</td>
<td>13.42</td>
</tr>
<tr>
<td></td>
<td>failures</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>$CUR2d_{CP}$</td>
<td>runtime</td>
<td>148.44</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>failures</td>
<td>5828091</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$CUR1d + CUR2d_{ILP}$</td>
<td>runtime</td>
<td>0.43</td>
<td>3.14</td>
<td>1.28</td>
</tr>
<tr>
<td></td>
<td>failures</td>
<td>235</td>
<td>6068</td>
<td>11452</td>
</tr>
<tr>
<td>$CUR1d + CUR2d_{CP}$</td>
<td>runtime</td>
<td>23.09</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>failures</td>
<td>302755</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 4.19: Finding an optimal solution. N/A stands for not available while a “-” means that no results were found within 1 clock hour.

4.4.5 Models based on set variables

The BACDP can also be viewed as a set partitioning problem, where the set of courses ought to be partitioned into $n$ subsets, one for each period. With this view on the BACDP, the model shown in Figure 4.21 declares an array $S$ of set variables that are subsets of the set of courses. To ensure that every course is given in exactly 1 period, the set variables must all be pairwisely disjoint and their union must equal the set of courses. To enforce that each period must cover at least 1 course, the cardinality of each set variable
<table>
<thead>
<tr>
<th>Model</th>
<th>Results</th>
<th>8 periods</th>
<th>10 periods</th>
<th>12 periods</th>
</tr>
</thead>
<tbody>
<tr>
<td>CUR1d</td>
<td>runtime failures</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CUR2dILP</td>
<td>runtime failures</td>
<td>1.29</td>
<td>3.45</td>
<td>13.42</td>
</tr>
<tr>
<td>CUR2dCP</td>
<td>runtime failures</td>
<td>-</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>CUR1d + CUR2dILP</td>
<td>runtime failures</td>
<td>0.43</td>
<td>3.14</td>
<td>1.28</td>
</tr>
<tr>
<td>CUR1d + CUR2dCP</td>
<td>runtime failures</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 4.20: Proving optimality. N/A stands for not available while a “-” means that no results were found within 1 clock hour.

is restricted to be greater than or equal to 1. The availability of global constraints on set variables, such as the (weighted-)cardinality constraints, in most constraint programming languages supporting set variables (CONJUNTO, OZ, ILOG SOLVER), makes it possible to compute the load of each period, as well as to state the restrictions on the amount of courses per period. However, with set variables, an inefficient formulation of the prerequisite constraints is unavoidable. We have to introduce more Boolean variables $B[i, j]$ and more constraints of the form:

$$B[i, j] = 1 \iff i \in S[j]$$

and state the prerequisites constraints in similar ways as in model CUR2dCP. Therefore, one should also consider channelling the model based on set variables into the model CUR1d in which the prerequisites constraints are easier to specify and reason about. Note also that the constraints

$$\forall i \in \text{periods} \cdot |S[i]| \geq 1$$

are redundant with the constraints that restrict the amount of courses of each period and will not prune any further inconsistent values. Thus, removing these redundant constraints will improve the proposed model.

### 4.4.6 A model using function variables

Our final model of the BACDP is an function model shown in Figure 4.22. The variables are two function variables. The LOAD function assigns to every period an integer academic load. The CUR function is from the set of courses into the set of periods. The objective function minimizes the maximum element of the range of the LOAD function. The load of any period $j$ is expressed as the sum of the credits of every course $i$ that has $j$ as an image under CUR. If course $i$ is a prerequisite of course $j$, then we enforce a strict ordering on their corresponding images. The academic load constraint is expressed as a set of inequalities. Finally, the number of courses of each period $j$ is captured by the cardinality of the set of courses that have $j$ as an image under CUR, which is restricted to be between the minimum and maximum numbers of allowed courses. Compared to the
Figure 4.21: A model of the BACDP based on set variables

previous models, all the constraints are expressed in the \( F \) model in a straightforward manner and there is a close correspondence between the given problem description in informal language and its \( F \) formulation. However, one major difference is that while \( L \) models can be made executable in a straightforward way, more work needs to be done in order to make the function models executable.

4.5 Summary

We considered models of three CSPs expressed in \( F \) and \( L \). We have shown in the case of the GCP that when we consider the optimization version, updating the \( L \) models was harder than updating the function model. The second example — on the WLP — compared the maintenance of the \( F \) and \( L \) models when we consider a minor change in the requirements. Again, it was straightforward to reflect the changes in the \( F \) model. However, it was not easy to reflect the changes in the \( L \) model in Figure 4.9, and we had to introduce more variables and reformulate some of the constraints in the model in Figure 4.10. In the last example (the BACDP), our concern was to develop an \( L \) model that efficiently solves all three real-life instances proposed in [11]. Even though we showed
that the $\mathcal{F}$ model is a much more natural formulation than the combined $\mathcal{L}$ models (that efficiently solved the problem), one important difference remains between $\mathcal{F}$ and $\mathcal{L}$. The $\mathcal{L}$ models can be made executable in a straightforward manner while the function models cannot.

Concerns about the solving time also require trade-offs about the level of abstraction for modelling: the function models expressed in $\mathcal{F}$ must after all be executable and should ideally execute quickly (and finitely). For instance, set constraint languages may well allow the formulation of constraints over sets (such as CLPS [2], CONJUNTO [39], NP-SPEC [8], OZ [55], and {LOG} [21]), hence providing a high level of abstraction for modelling, but if they cannot be compiled into acceptably fast code, then the advantage of decreased programming time is neutralized by the disadvantage of increased solving time. The same holds for $\mathcal{F}$, which leads us to the following questions:

- How can we translate function models written in $\mathcal{F}$ into executable models?
- How do the generated models resulting from the function models compare to models expressed in $\mathcal{L}$? Will they be more efficient or less efficient?
- Quite often alternate models are tried in $\mathcal{L}$ in order to find a reasonably efficient model. Will the translation of the function models generate one corresponding model or many?
- If the translation of function models generates alternate models, then which model to choose, and why?

Addressing these questions is important and will be the concern of the following chapters.
Chapter 5

Alternate Model Generation

In Chapter 4 we argued that having function variables and constraints on these variables makes the modelling of some CSPs at a high level of abstraction. The resulting function models are easier to understand and to maintain. The major reason for achieving high-level models is to decrease the development and maintenance costs as well as to make CP accessible to a wider range of users. However, these high-level models need to be made executable and “fast”.

This chapter is organized as follows. We show in Section 5.1 that a function variable can be represented in many different ways in terms of (arrays of) integer and set variables. Depending on the choice of representation for a function variable, the constraints are translated differently. We also show that sequences and permutations can be viewed as functions that satisfy extra properties. In Section 5.2, we show how to integrate different representations of function variables while we show some useful implied constraints in Section 5.3. Finally, we summarize and state the challenges to be addressed in the following chapters in Section 5.4.

5.1 Rewriting an \( F \) model in terms of alternate \( L \) models

We here show how to represent function, sequence, and permutation variables in terms of integer and set variables. We also show how to rewrite constraints on the new variables.

5.1.1 Functions and their operations

In Chapter 4 we have shown that in some CSPs the desired outputs can be captured as function variables. In the following, we will show some possible representations of function variables and their allowed operations in terms of (arrays of) integer and set variables, and constraints on variables of these types.

Total function representations. At a lower level of abstraction, namely the CP level, a function variable can be represented in many different ways. For instance, three equivalent ways of representing a total function variable \( F \) from \( \{1, 2, 3\} \) into \( \{1, 2, 3, 4\} \) are shown in Figure 5.1.

In Figure 5.1 and subsequent figures in this section, the parts of a function model in consideration are enclosed in a dashed rectangle. An arrow is used with an annotating
number to point to possible \( L \) variables and constraints that represent the \( F \) parts. At the bottom of each alternative, we relate the function variable with the \( L \) variables by providing the meaning of the assignment of a value to a variable at the \( L \) level of abstraction in terms of membership in the function variable at the \( F \) level of abstraction. The notation \( Fd1 \) stands for a one-dimensional array (of integer variables) representation for a function variable \( F \), while \( Fd2 \) stands for a two-dimensional Boolean array representation for a function variable \( F \), and \( S \) stands for a one-dimensional array (of set variables) representation for a function variable \( F \).

We abuse our notation and refer to the \( L \) variables and constraints that use \( Fd1 \) as \( Fd1 \). Similarly, we refer to the \( L \) variables and constraints that use \( Fd2 \) (resp. \( S \)) as \( Fd2 \) (resp. \( S \)).

Given a function variable declaration \( F : V \rightarrow W \), the first representation uses a 1d array \( Fd1 \) of integer variables. The array \( Fd1 \) is indexed by \( V \) and ranges over \( W \). The first representation \( Fd1 \) encodes the function variable \( F \) only under the following meaning:

\[
\forall i \in V \cdot \forall j \in W \cdot Fd1[i] = j \iff \langle i, j \rangle \in F
\]

We use \( \iff \) instead of \( \leftrightarrow \) because we are relating two entities that are conceptually at different levels of abstractions.

The second representation \( Fd2 \) encodes the function variable \( F \) as a 2d array \( Fd2 \) of Boolean variables. The array \( Fd2 \) is indexed by \( V \) and \( W \). In addition, each row sum must be equal to 1 if we want to associate the following meaning to the Boolean variables:

\[
\forall i \in V \cdot \forall j \in W \cdot Fd2[i, j] = 1 \iff \langle i, j \rangle \in F
\]

The third representation uses an array \( S \) of set variables, indexed by \( W \), and the set variables are subsets of \( V \). If we want to associate the meaning:

\[
\forall i \in V \cdot \forall j \in W \cdot i \in S[j] \iff \langle i, j \rangle \in F
\]

Figure 5.1: Alternative \( L \) representations of a total function variable
then we must guarantee that each element in $V$ belongs to exactly one set variable, and that all elements of $V$ are covered by the set variables. Thus, the set variables are constrained to be pairwisely disjoint and their union is constrained to be equal to the set $V$.

We consider the case when $F : V \rightarrow \text{int}$ as a special case, and consider the first representation as the only alternative. The reason is that the cardinality of the source set $W$ is very large ($2 \times \text{maxint} + 1$) so it becomes impractical to have a 2d array with one dimension that large as well as having that many set variables.

Partial function representations. If we declare a function variable as $F : V \mapsto W$, then not all elements of $V$ have an image under $F$. One possible way to overcome this is to introduce a dummy element to $W$ and view the partial function as a total one instead, where all the elements in $V$ that have no image under the desired partial function $F$ are mapped to the dummy value. However, this requires rethinking the problem constraints and making sure that they hold only when the image of an element of $V$ is not the dummy value. This transformation cannot be done in a systematic way because the elements of $W$ may have certain semantics and adding a dummy variable cannot be random then. For instance, suppose each element in $W$ has an associated weight that is used to state the problem constraints, then what weight shall we associate to the dummy element? Answering this question is problem dependent.

So, is it possible instead to relax some of the constraints posed on $F_d1, F_d2,$ and $S$ for the case of a total function? Three different ways of modifying the representation $F_d1, F_d2,$ and $S$ in Figure 5.1 are shown in Figure 5.2. The three representations are modified by introducing extra variables to explicitly represent the domain of the partial function $F$. We refer to the domain of $F$ as $\text{dom}_F$ in all representations. In $F_d1$ and $F_d2$ a 1d array $\text{dom}_F$ of Boolean variables, indexed by $V$, is introduced, while a set variable $\text{dom}_F$ that is a subset of $V$ is introduced in the representation $S$.

In the representation $F_d1$, we find it difficult to link the two arrays $\text{dom}_F$ and $F_d1$. But the chosen representation captures the desired partial function iff $F_d1[i] = j \land \text{dom}_F[i] = 1$ represents $\langle i, j \rangle \in F$.

Unlike the representation $F_d1$, the variables in $F_d2$ can be linked to $\text{dom}_F$. For each row $i$, the sum is constrained to be equal to $\text{dom}_F[i]$. Thus making sure that if an element $i$ is not in $\text{dom}_F$ (when $\text{dom}_F[i] = 0$) then its corresponding row is set to zero and vice versa, which allows $F_d2[i, j] = 1$ to represent $\langle i, j \rangle \in F$.

Similarly to $F_d2$, the set variables in $S$ can be related to the set variable $\text{dom}_F$ by constraining the union of the set variables $S[j]$ to be equal to $\text{dom}_F$, thus, making sure that if an element $i$ is not in $\text{dom}_F$ (when $i \notin \text{dom}_F$), then it is not an element of any set variable and vice versa, which allows $i \in S[j]$ to represent $\langle i, j \rangle \in F$.

Injection representations. Assume that a desired total function must be injective. Figure 5.3 shows the formulation of the injectiveness constraint on the three $L$ representations $F_d1, F_d2,$ and $S$ of total functions.

For $F_d1$, we show three ways of enforcing that all variables in the array $F_d1$ have distinct values. More alternatives will be presented in Chapter 6. Alternative $a$ poses binary not-equal constraints between every pair of variables. Alternative $b$ uses the global $\text{alldifferent}$ constraint. The third alternative, $c$, introduces a redundant array of variables.
\(DFd1\), representing the partial inverse function \(F^{-1}\), where the values of \(Fd1\) are exchanged for variables and vice versa. Enforcing that the variables in \(Fd1\) have distinct values is achieved through channelling constraints of the form \(Fd1[i] = j \rightarrow DFd1[j] = i\). The proof that these channelling constraints enforce the injectiveness will be presented in Chapter 6.

Column-sum constraints are used in \(Fd2\) while cardinality constraints are used for the set variables representation to enforce the injectiveness. Now, if the desired injection is partial instead, we show the formulation of the injectiveness constraint on each representation of the partial function in Figure 5.4. When using a 1d array \(Fd1\) together with the 1d 0/1 array \(dom_F\), we state a clique of quaternary constraints, which say that if two elements belong to the domain of the function, then their images must be different. For \(Fd2\), we add column-sum constraints while we constrain the cardinality of each set variable \(S[j]\) in the third representation \(S\).

**Surjection representations.** When a desired total function variable must be surjective, either the global constraint \(\text{atleast}\) can be used on the first representation (alternative \(a\) in Figure 5.5), or a set of Boolean variables \(B_{ij}\) is introduced and constrained as follows:

\[ B_{ij} = 1 \iff Fd1[i] = j \]

which allows the formulation of the surjectiveness constraint as a set of summation constraints on the Boolean variables \(B_{ij}\), as shown in alternative \(b\) in Figure 5.5. Column-sum constraints may be used for \(Fd2\), while for \(S\) we constrain every set variable \(S[j]\) to contain at least one element as shown in Figure 5.5. Note that the set variables and their constraints model the surjective function as a set partitioning problem.

In the case of the surjection \(F\) being partial, the formulation of the surjectiveness constraint on each representation is shown in Figure 5.6. For \(Fd1\), we introduce Boolean
variables $B_{ij}$ and constrain each of them as follows:

$$B_{ij} = 1 \iff (\text{dom}_F[i] = 1 \land Fd1[i] = j)$$
Figure 5.5: Alternative representations of a surjective total function variable

which allows the formulation of the surjectiveness constraint as a set of summation constraints on these Booleans. For the second representation, $Fd_2$, we enforce the surjectiveness constraint as column-sum constraints, while for $S$ we constrain the cardinality of the set variables $S[j]$.

**Bijection representations.** In Figure 5.7, alternatives of enforcing that a desired total function is bijective are presented.

Alternative $a$ is conceptually the result of combining alternative $a$ in Figure 5.3 and alternative $a$ or $b$ in Figure 5.5 because a bijective function is both injective and surjective. However, since for total bijective functions we have $|V| = |W|$, alternative $a$ and $b$ in Figure 5.5 are implied by alternative $a$ in Figure 5.3 and thus are redundant. Therefore we only use alternative $a$ in Figure 5.3 for the bijective functions. Similarly, alternative $b$ in Figure 5.7 is conceptually the result of combining alternative $b$ in Figure 5.3 and alternative $a$ or $b$ in Figure 5.5. But we only use alternative $b$ in Figure 5.3 for the bijective functions because both alternative $a$ and $b$ in Figure 5.5 are redundant when $|V| = |W|$. Alternative $c$ in Figure 5.7 is the result of strengthening alternative $c$ in Figure 5.3 because the inverse of a bijective function is also a bijective function, so the other direction of the implication in alternative $c$ in Figure 5.3 holds for the bijective case.

For $Fd_2$ and $S$, we combine and simplify both formulations in Figure 5.3 and in Figure 5.5. The simplification rule is as follows:

$$(a \leq 1 \land a \geq 1) \iff a = 1$$

The case when a desired partial function is bijective is shown in Figure 5.8. For $Fd_1$, the chosen formulation is the result of strengthening the formulation for $Fd_1$ in Figure 5.4 and the translation of the extra constraint $\text{domain}(F) = |W|$, which makes the formulation for
Figure 5.6: Alternative representations of a surjective partial function variable

Figure 5.7: Alternative representations of a bijective total function variable

$Fd1$ in Figure 5.6 redundant. For $Fd2$ and $S$, we combine and simplify both formulations in Figure 5.4 and in Figure 5.6.
Representing function operations. As demonstrated earlier, there are different ways of representing function variables in terms of integer and set variables along with constraints on these variables. Our attention is now focused on the translation of the function operations into expressions and constraints in terms of the integer and set variables used to encode a function variable. We distinguish between two major cases, namely the case when the desired function is total and the case when it is partial. We assume that we are looking for a function \( F \) from set \( V \) into set \( W \). We here summarize the different ways of encoding the different cases of the desired function in terms of integer variables, ignoring the extra constraints:

\[
\begin{array}{|c|c|}
\hline
\text{1d array} & \text{2d 0/1 array} \\
\hline
F : V \rightarrow W & Fd1[V] \in W \quad Fd2[V,W] \in 0..1 \\
F : V \mapsto W & Fd1[V] \in W \quad Fd2[V,W] \in 0..1 \\
\text{dom}_F[V] \in 0..1 & \text{dom}_F[V] \in 0..1 \\
\hline
\end{array}
\]

The different representations of the cases of a function in terms of set variables are as follows, again ignoring the extra constraints:

\[
\begin{array}{|c|c|}
\hline
\text{Set variables} & \text{Membership in functions.} \\
\hline
F : V \rightarrow W & S[W] \subseteq V \\
F : V \mapsto W & S[W] \subseteq V \\
\text{dom}_F \subseteq V & \text{dom}_F \subseteq V \\
\hline
\end{array}
\]

The membership predicate \( \langle i, j \rangle \in F \) can be encoded differently depending on our choice of representation for the function variable \( F \).
If $F$ is a total function and the chosen representation is $Fd1$, then $\langle i, j \rangle \in F$ is represented as $Fd1[i] = j$. When $F$ is partial and the chosen representation is $Fd1$ and $\text{dom}_F$, then the membership in function is represented as $Fd1[i] = j \land \text{dom}_F[i] = 1$.

Regardless of whether the function is partial or total, the membership predicate is always translated into $Fd2[i, j] = 1$ when $Fd2$ is the representation choice. Similarly, when set variables are used to represent the function, the membership predicate is $i \in S[j]$.

The membership predicate may appear in the arithmetic aggregate operator $\sum$, as follows:

$$\sum_{\langle i, j \rangle \in F} Q(i, j)$$

where $Q(i, j)$ is any arithmetic expression. When the first representation ($Fd1$) is used, the summation expression is translated into:

$$\sum_{i \in V} Q(i, Fd1[i])$$

when the function $F$ is total, and into:

$$\sum_{i \in V} Q(i, Fd1[i]) \ast \text{dom}_F[i]$$

when the function $F$ is partial. When the second representation ($Fd2$) is used, regardless of whether the function $F$ is total or partial, we have:

$$\sum_{i \in V} \sum_{j \in W} Fd2[i, j] \ast Q(i, j)$$

Finally, when the third representation ($S$) is used, regardless of whether the function $F$ is total or partial, we have:

$$\sum_{i \in V} \sum_{j \in W} B_{i,j} \ast Q(i, j)$$

where $B_{i,j} \in 0..1$ are new variables and $B_{i,j} = 1 \iff i \in S[j]$, for all $i \in V$ and $j \in W$.

It may also be the case that the membership predicate participates in the logic quantifier \(\forall\), as follows:

$$\forall \langle i, j \rangle \in F \cdot P(i, j)$$

where $P(i, j)$ is any logical formula. When the first representation ($Fd1$) is used, the formula will be rewritten into

$$\forall i \in V \cdot P(i, Fd1[i])$$

when the function $F$ is total, and

$$\forall i \in V \cdot \text{dom}_F[i] = 1 \rightarrow P(i, Fd1[i])$$

when the function $F$ is partial. When the second representation ($Fd2$) is used, regardless of whether the function $F$ is total or partial, we have:

$$\forall i \in V \cdot \forall j \in W \cdot Fd2[i, j] = 1 \rightarrow P(i, j)$$

Finally, when the third representation ($S$) is used, regardless of whether the function $F$ is total or partial, we have:

$$\forall i \in V \cdot \forall j \in W \cdot i \in S[j] \rightarrow P(i, j)$$
**Function application.** Retrieving the image of an element \( i \in \text{domain}(F) \) under \( F \) (denoted by \( F(i) \)) can be translated to different expressions on the integer and set variables. When \( F \) is either total or partial and the representation choice is \( Fd1 \), we have

\[
Fd1[i]
\]

However, when \( Fd2 \) is used, retrieving the image of \( i \in \text{domain}(F) \) under \( F \) can be mapped to the sum expression:

\[
\sum_{j \in W} Fd2[i, j] \cdot j
\]

In the case when \( S \) is used to represent the desired function, the situation is a little bit more complex. Retrieving the image requires us first to introduce new Boolean variables \( B_{i,j} \) and add the following constraints:

\[
\forall i \in V \cdot \forall j \in W \cdot B_{i,j} \in \{0, 1\}
\]

\[
\forall i \in V \cdot \forall j \in W \cdot i \in S[j] \iff B_{i,j} = 1
\]

so that \( F(i) \) corresponds to the summation expression:

\[
\sum_{j \in W} B_{i,j} \cdot j
\]

**Inverse function application.** Retrieving all the elements \( i \in \text{domain}(F) \) that have image \( j \in W \) is denoted by \( F^{-1}(j) \). Conceptually, \( F^{-1}(j) \) is a set, which is explicitly declared in the representation \( S \). Each set variable \( S[j] \) denotes the set of elements that have image \( j \) in \( W \). In \( Fd2 \), such sets also are explicitly declared as the columns of the 2d array \( Fd2 \). Note that having an array of Boolean variables is just an alternative way of encoding a set variable. Given a column \( j \), for all \( i \in V \), we have that \( Fd2[i, j] = 1 \) iff \( i \in F^{-1}(j) \).

In contrast, the set of such elements is implicit in the 1d array \( Fd1 \) that encodes a total function. Therefore, more variables are needed to explicitly represent such sets. We may introduce \( n \) Boolean variables \( B_{i,j} \) for each element \( j \in W \), where \( n \) is the cardinality of \( V \). For each \( j \), we then have to add the following channelling constraints:

\[
\forall i \in V \cdot Fd1[i] = j \iff B_{i,j} = 1
\]

Now, for each \( j \), the \( n \) Booleans represent the elements \( i \in V \) that have \( j \) as image under \( F \), under the meaning that \( B_{i,j} = 1 \) iff \( i \in F^{-1}(j) \). When the function is partial, we need to add the following channelling constraints instead:

\[
\forall i \in V \cdot \text{dom}_F[i] = 1 \land Fd1[i] = j \iff B_{i,j}
\]

Operations on \( F^{-1}(j) \) get rewritten into different constraints/expressions depending on the choice of representation. Given a set \( T \), a set \( S \subseteq T \) can be represented either using Boolean variables, encoded as a 0/1 array \( S_b \) indexed by \( T \), or using a set variable, which we call \( S_v \), based on the interval representation proposed in [39]. The building blocks for the translation of expressions and constraints on \( S \) are presented in Figure 5.9.

The element \( i \) is an element of \( S \) is represented as \( S_b[i] = 1 \) in the Boolean representation and as \( i \in S_v \) in the interval representation. The cardinality of set \( S \) is represented
as a summation expression on the Booleans while a global constraint can be stated on $S_v$.

The expression:

$$\sum_{i \in S} Q(i)$$

is rewritten as a weighted-sum expression over the Booleans $S_b[i]$ while a global constraint is stated on $S_v$. Finally, stating a constraint $P(i)$ over the elements of $S$ is rewritten as a set of implication constraints of the form

$$S_b[i] = 1 \rightarrow P(i)$$
on the Booleans and of this form

$$i \in S_v \rightarrow P(i)$$
on $S_v$.

**Function domain.** The domain of a total function $F : V \rightarrow W$ is the set $V$. When $F$ is partial, its domain is represented as a Boolean array $dom_F$ in the representations based on integer variables ($Fd_1$ and $Fd_2$), and as a set variable (also named $dom_F$) in the set variable representation $S$.

**Function range.** When a partial or total function $F$ from set $V$ into set $W$ is surjective or bijective, then the range of $F$ is $W$. Otherwise, the range of $F$ is *implicit* in all three representations.

In the first and second representations, the range can be defined as a new 1d 0/1 array $range_F$ indexed by $W$, with $range_F[j] = 1$ if $j \in range(F)$. Therefore, we need to link the Boolean variables $range_F[j]$ with the already existing variables. When $Fd1$ is used to represent a total function, we may add the following linking constraints:

$$\forall i \in V \cdot \forall j \in W \cdot Fd1[i] = j \rightarrow range_F[j] = 1$$

which express that whenever $j$ gets mapped to an element $i$, then it must be in the range. If both arrays $Fd1$ and $dom_F$ are used, i.e., when $F$ is partial, we pose instead the following linking constraints:

$$\forall i \in V \cdot \forall j \in W \cdot (dom_F[i] = 1 \land Fd1[i] = j) \rightarrow range_F[j] = 1$$

which makes an extra check to make sure that $i$ belongs to the domain of $F$.

In the case of the 2d 0/1 array $Fd2$, we add the following constraints for both partial and total functions:

$$\forall j \in W \cdot \forall i \in V \cdot Fd2[i, j] \leq range_F[j]$$

67
which sets every column of array $Fd2$ to 0 whenever $j$ is not in the range of $F$.

When the representation $S$ is used, we may explicitly declare the range of $F$ as a new set variable $range_F \subseteq W$. But we have to link $range_F$ to the set variables $S[j]$ as follows:

$$\forall j \in W \cdot |S[j]| \geq 1 \rightarrow j \in range_F$$

which enforces that if set variable $S[j]$ — representing all the elements that have image $j$ — has at least one element, then $j$ must belong to the range of $F$.

However, when we have the expression $\text{max}(\text{range}(F))$ (resp. $\text{min}(\text{range}(F))$) in a function model, then no extra variables are needed as we can use the global constraint $\text{max}(F1d)$ (resp. $\text{min}(F1d)$).

5.1.2 Sequences and their operations

In many CSPs, the objective is to find a sequence over a given set. For instance, in the magic sequence problem (Prob019 at www.csplib.org), the objective is to find a sequence $S$ of length $n$ over the set $\{0, \ldots, n-1\}$, such that for all $i \in \{0, \ldots, n-1\}$, the number $i$ occurs $S(i)$ times. Another example is the car sequencing problem (Prob001 at www.csplib.org), where the objective is to arrange a set of cars into a sequence such that the capacity of each station is not violated.

Sequences of fixed or bounded size can be viewed as functions. A sequence of fixed size corresponds to a total function, whereas a sequence of bounded size corresponds to a partial function. Figure 5.10 shows the correspondence between sequences and functions. A sequence $S$ of fixed size $k$ over a set $W$ is a function $S : V \rightarrow W$, where $V = \{1, \ldots, k\}$. A sequence $S$ of bounded size $k$ over a set $W$ is a partial function from $\{1, \ldots, k\}$ into $W$, but the elements of the domain of the partial function must be a prefix of the integer range 1..k. As for the operations on sequences, there thus exists a one-to-one correspondence with function operations. Retrieving the $i^{th}$ element of a sequence corresponds to function application for element $i$. The domain and range of a sequence correspond to the domain
and range of the corresponding function, respectively. Finally, the length of the sequence is the cardinality of the domain of the corresponding function. Representing fixed-sized sequences is equivalent to representing total functions.

However, for the case of bounded-sized sequences, the corresponding partial function must satisfy an extra condition, namely that the domain of the corresponding partial function must be a prefix of 1..k. Considering the different ways of representing partial functions in Figure 5.2, we present for each representation an extra constraint to enforce that the partial function does actually represent a sequence of bounded size. In the first and second representation, where \( \text{dom}_F \) is used to represent the elements of the domain, we enforce a non-decreasing ordering on the Booleans:

\[
\forall i \in 1..k - 1 \cdot \text{dom}_F[i] \geq \text{dom}_F[i + 1]
\]

However, in the set variable representation, we need to enforce that the set variable \( \text{dom}_F \) can only take a value from a selected subset of the set of all possible subsets:

\[
\text{dom}_F \in \{\{\}, \{1\}, \{1, 2\}, \ldots, \{1, \ldots, k\}\}
\]

This selected subset of the set of all possible subsets has only those that represent prefixes of 1..k.

### 5.1.3 Permutations and their operations

In many CSPs, the objective to find a permutation over a given set. Many examples are discussed in [67, 66, 81].

A permutation over a set \( W \) is a sequence that has exactly the elements of \( W \), which corresponds to a total bijection from \( V = \{1, \ldots, |W|\} \) into \( W \). Figure 5.11 shows the correspondence between permutations and functions. The permutation operations are the same as the sequence ones, and since we represent a sequence of fixed size as a total function, we represent permutations the same way as for total functions, but we add the bijectiveness constraint.

### 5.1.4 Limitations

There are limitations to our approach in terms of the type of expressions that we are able to rewrite from \( \mathcal{F} \) to \( \mathcal{L} \). Arbitrarily nested expressions such as \( |\text{range}(F) \cap \text{range}(G)| \) are not properly handled by our approach. To overcome this one should use higher-order facilities such as \( \lambda \)-abstractions. However, as will be demonstrated in Chapter 7, in many practical applications such types of expressions do not arise, and thus our approach is still useful.
5.2 Integration and channelling constraints

We first motivate the need for integrating different representations and then present our approach to integration.

5.2.1 Benefits of model integration

In alternative c in Figure 5.3 and Figure 5.7, a dual representation $DFd1$, representing the inverse function, of the array $Fd1$ where the values are exchanged for the variables is used along with channelling constraints to enforce the bijectiveness and injectiveness constraints of a total function. The introduction of the redundant variables and the channelling constraints is shown to be beneficial in the case of total bijections [81]. A similar analysis for injective total functions is carried out in Chapter 6 and it is also shown to be beneficial to have redundant variables and channelling constraints. This idea of “redundant modelling” can be generalized to integrate different representations of a problem.

The integration of different representations of a problem has been studied by Cheng et al. [12], Smith [66], and Walsh [81], for instance. A similar idea was previously suggested by Geelen [36]. There are a number of benefits of integration as shown in [23]:

- **Improved propagation**: By integrating different models, the amount of propagation carried out in each model may be significantly improved, giving a more powerful model than any of the participating models. For instance, the steel mill slab design problem [34] is a bin packing problem with a side constraint restricting the number of “colors” of orders placed on any slab. This side constraint is easily implemented by channelling into a 2d array model, which improves the amount of propagation. More examples can also be found in [23].

- **Ease of statement**: It may also be the case that when some constraints are difficult to specify and reason about in a particular problem representation, one should consider channelling into a second model in which these constraints are easier to specify and reason about. For instance, in the model $CUR1d + CUR2dCP$ and the model $CUR1d + CUR2dILP$ of the BACDP in Chapter 4, some of the constraints are easier to specify and reason about in some of the participating models.

- **Cheap implementation**: Another potential for integration may be seen in conjunction with variable and value ordering heuristics, that is integrating different models may allow us to develop cheap value ordering heuristics when the values are explicitly represented in one of the participating models. One may as well branch on the whole set of variables at the same time, as shown in [66].

- **Symmetry breaking**: Some standard methods may be employed to deal with symmetrical variables in the models such as row and column symmetry in matrix models [24]. These same techniques may also be applied to deal with indistinguishable values when the values are explicitly represented as variables in another model. For instance, in the rack configuration problem [48] such techniques may be employed. More examples are also discussed in [48].

We will now show our approach to integrating the $L$ models resulting from a function model.
5.2.2 Our approach to automatic model integration

We distinguish between two types of model integration; the first type integrates two models where the solution method to be employed is the same, while the second type is when different solution methods are used for each of the participating models. In particular, we restrict ourselves to two solution methods, which are CP and ILP. We will use ILP techniques to solve models of CSPs where all the constraints are linear. We will refer to models of the first type of integration as integrated models while models of the second type are referred to as hybrid models. Furthermore, if all the constraints from both models are used then we have full integrated/hybrid models, otherwise partial integrated/hybrid models.

In Section 4.4, the model \( CUR1d + CUR2d_{ILP} \) is a partial hybrid model while the model \( CUR1d + CUR2d_{CP} \) is a partial integrated model. As an example of a full hybrid model, let’s consider the optimization version of the WLP presented in Section 4.3. The \( L \) model in Figure 4.9 and the \( L \) model in Figure 4.10 can be combined into a full hybrid model as advocated in [75]. Note that the constraints:

\[
\forall i \in \text{stores} \cdot \forall j \in \text{warehouses} \cdot \text{SUPPLIER}[i, j] = 1 \rightarrow \text{OPENW}[j] = 1
\]

in the model in Figure 4.10 are rewritten as linear constraints as follows:

\[
\forall i \in \text{stores} \cdot \forall j \in \text{warehouses} \cdot \text{SUPPLIER}[i, j] \leq \text{OPENW}[j]
\]

thus making all the constraints of the model in Figure 4.10 linear.

In what follows we make to the following restrictions:

- We take a conservative approach and consider integrated/hybrid models of problems expressed in \( \mathcal{F} \) where we have at least one total function variable. We will not consider integrating different representations of partial functions. Furthermore, we only consider combining any two of the three representations of each function variable (\( Fd_1, Fd_2, \) and \( S \)).

- Integrated models will use only CP as a solution method when solved on particular instances.

- Hybrid models are only considered for optimization problems. The objective function is then the one of the model that uses an ILP solution method to solve particular instances.

- Only models of function problems that use the representation \( Fd_2 \) may use an ILP solver to solve particular instances.

The representations \( Fd_1, Fd_2, \) and \( S \) of a function variable that is total differ in many aspects. They differ in the type of variables used, namely from integer variables of Boolean domains to integer variables of larger domains to set variables. The representations also differ in the number of variables, the number of constraints, and the type of constraints. Some of the constraints are global while others are linear for instance.

Our approach is to analyze the possible formulations in a general way. We will compare them whenever it is possible and incorporate the lessons learnt into heuristics that recommend which formulation to pick as the integrated/hybrid model formulation.
Arithmetic constraints involving function application. For function problems, some of the commonly posted constraints are arithmetic ones involving function applications. For instance, the constraint in the GCP (Section 4.2):

$$\forall \langle i, j \rangle \in \text{edges} \cdot \text{COLORING}(i) \neq \text{COLORING}(j)$$

and the prerequisite constraint in the BACDP (Section 4.4):

$$\forall \langle i, j \rangle \in \text{prereq} \cdot \text{CUR}(i) < \text{CUR}(j)$$

are such kinds of constraints.

However, depending on the choice of the representation of the function variable, these constraints get mapped to constraints that have different characteristics. Let us take the constraint $F(i) < F(j)$ as a running example where $F$ is declared as $F : V \rightarrow W$.

In $Fd_1$, we have that $F(i)$ gets mapped to $Fd_1[i]$, thus the corresponding constraint would be

$$Fd_1[i] < Fd_1[j]$$

In $Fd_2$, we have that $F(i)$ corresponds to $\sum_{k \in W} Fd_2[i, k] \cdot k$, thus $F(i) < F(j)$ corresponds to:

$$\sum_{k \in W} Fd_2[i, k] \cdot k < \sum_{k \in W} Fd_2[j, k] \cdot k$$

Finally, in the $S$ encoding, $F(i)$ is mapped to $\sum_{k \in W} B_{i,k} \cdot k$ where

$$\forall k \in W : B_{i,k} \in \{0, 1\}$$

$$\forall k \in W : i \in S[j] \iff B_{i,k} = 1$$

Thus, $F(i) < F(j)$ is translated into:

$$\sum_{k \in W} B_{i,k} \cdot k < \sum_{k \in W} B_{j,k} \cdot k$$

Analyzing these formulations of $F(i) < F(j)$ leads us to the following:

- In $Fd_1$, we have a binary constraint. In general, if we have an arithmetic constraint on a function variable that uses $n$ function applications, then the corresponding translation into the first encoding will yield a constraint of arity $n$ with variables of domain size $|V|$.

- In $Fd_2$, we have a constraint of arity $2 \cdot |W|$. In general, if we have an arithmetic constraint on a function variable that uses $n$ function applications, then the corresponding translation into the second encoding will yield a constraint of arity $n \cdot |W|$ with variables of domain size 2.

- In $S$, we introduce $2 \cdot |W|$ extra Boolean variables, employ $2 \cdot |W|$ extra constraints of arity 2, and state a constraint of arity $2 \cdot |W|$. In general, if we have an arithmetic constraint on a function variable that uses $n$ function applications, then the corresponding translation introduces $n \cdot |W|$ extra Boolean variables, $n \cdot |W|$ extra constraints of arity 2, and a constraint of arity $n \cdot |W|$ with variables of domain size 2.
The different formulations express differently arithmetic constraints on functions. However, the amount of pruning achieved in either of them is dependent on the level of consistency enforced by a particular solver on the different variations of the constraints.

The question now is if we integrate any two of the representations of a function variable, then which arithmetic constraint formulation to use? We will use the following criteria which will guide us in developing a heuristic that recommends which formulation is to be used:

- **Constraint number**: it is better to have as few constraints as possible.

- **Constraint arity**: the lower the constraint arity is, the better.

- **Constraint linearity**: linear constraints can be efficiently solved using an ILP solver while some non-linear constraints (e.g., inequality) are efficiently solved using CP techniques.

Now, given a problem formulation in $F$ where a total function $F$ is to be found such that some constraints are satisfied, there is a number of different ways of integrating any two of the representations that encode the function variables, taking into consideration the restrictions we set earlier:

- **Integrated model $Fd_1 + Fd_2$**: We combine $Fd_1$ and $Fd_2$ and use CP solver to solve instances of the resulting model. The $+$ symbol stands for combining two representations through channelling constraints that will be discussed later.

- **Integrated model $Fd_1 + S$**: We combine $Fd_1$ and $S$ and use a CP solver to solve instances of the resulting model.

- **Integrated model $Fd_2 + S$**: We combine $Fd_2$ and $S$ and use a CP solver to solve instances of the resulting model.

- **Hybrid model $Fd_1 + Fd_2$**: We combine $Fd_1$ and $Fd_2$ and solve instances of the resulting model by using a CP solver on the constraints of $Fd_1$ and an ILP solver on the constraints of $Fd_2$.

- **Hybrid model $Fd_2 + S$**: We combine $Fd_2$ and $S$ and solve instances of the resulting model by using a CP set-solver on the constraints of $S$ and an ILP solver on the constraints of $Fd_2$.

Based on the previous analysis, we propose the heuristic $H_{\text{arithmetic}}$ which decides which arithmetic constraint formulation to use. The heuristic $H_{\text{arithmetic}}$ for integrated models is shown in Figure 5.12 and the one for hybrid models is shown in Figure 5.13. The symbol $\sqrt{}$ marks the chosen constraint formulation. Any extra condition that must be satisfied by the selected constraint is given between parentheses. The integrated models $Fd_1 + Fd_2$ and $Fd_1 + S$ and the hybrid model $Fd_1 + Fd_2$ use the formulation on $Fd_1$ because the constraint formulation on this representation leads to constraints of lower arity. The integrated model $Fd_2 + S$ uses the formulation on $Fd_2$ because there are then fewer constraints. The hybrid model $Fd_2 + S$ uses the formulation on $Fd_2$ when the constraint is linear because there are then fewer constraints, and uses the formulation on $S$ when the constraint is non-linear because such constraints are delayed when ILP techniques are used.
Inverse function application constraints. When a function $F$ is declared as $F : V \rightarrow W$, the set of all elements that have image $j \in W$ under $F$ is denoted by $F^{-1}(j)$. This inverse function application appears in many useful constraints. For instance, in the WLP (in Section 4.3), the constraint that each warehouse has a capacity that should not be exceeded is expressed as follows:

$$\forall j \in \text{warehouses} \cdot |\text{SUPPLIER}^{-1}(j)| \leq \text{capacity}(j)$$

In the BACDP (in Section 4.4), to compute the load of each period $j$, we sum the credits of all courses that have $j$ as image under the $\text{CUR}$ function, as follows:

$$\forall j \in \text{periods} \cdot \text{LOAD}(j) = \sum_{i \in \text{CUR}^{-1}(j)} \text{credit}(i)$$

In particular, we consider two commonly posted constraints using inverse function application. It is quite frequent that the cardinality of $F^{-1}(j)$ is restricted. We refer to such constraints as capacity constraints, which have the following generic form:

$$|F^{-1}(j)| \text{ RelOp capacity}(j)$$

where $\text{RelOp}$ is $<, \leq, >, \geq, =$, and $\text{capacity}$ is a function returning the capacity of $j$. The capacity of each element in $F^{-1}(j)$ may have a weight in which case we have a weighted capacity constraint, which has the following generic form:

$$\left( \sum_{i \in F^{-1}j} \text{weight}(i) \right) \text{ RelOp capacity}(j)$$

where $\text{weight}$ is a function returning the weight of $i$.

If we have a capacity constraint, then a global constraint such as $\text{atleast}, \text{atmost}$, and $\text{exactly}$ (in Section 2.3) is employed in $Fd1$. It is also the case that global constraints on set variables such as the cardinality constraint and the weighted-cardinality constraint (in Section 2.3) are employed for the capacity constraints and the weighted capacity constraints, respectively. When $Fd2$ is the representation choice, the (weighted) capacity constraints are mapped into linear constraints as shown in Figure 5.9. Thus, for the capacity constraint, we use a global constraint in $Fd1$, a linear constraint in $Fd2$, and a global constraint is used in $S$. As for the weighted capacity constraint, we use a linear constraint in $Fd2$ and a global constraint in $S$.

<table>
<thead>
<tr>
<th>$F_d1 + F_d2$</th>
<th>$F_d2 + S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Figure 5.12: The heuristic $H_{\text{arithmetic}}$ for integrated models

<table>
<thead>
<tr>
<th>$F_d1 + F_d2$</th>
<th>$F_d2 + S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓ (linear constraint) ✓ (non-linear constraint)</td>
</tr>
</tbody>
</table>

Figure 5.13: The heuristic $H_{\text{arithmetic}}$ for hybrid models
In general now, when the constraints involving inverse function application operation are of neither of the special cases discussed above, the set $F^{-1}(j)$ is explicitly represented in $F_d2$ as the column $j$ of Boolean variables, and in $S$ as the set variable $S[j]$. However, we introduce more Boolean variables and constraints in order to represent $F^{-1}(j)$ when $F_d1$ is the representation choice for the function $F$. The introduction of such Boolean variables is analogous to the columns of the 2d 0/1 array in $F_d2$. Therefore, we will have the same constraint formulation for constraints involving $F^{-1}(j)$ in $F_d1$ and $F_d2$. However, in the first representation we will have the burden of having extra (Boolean) variables and extra constraints.

Based on this analysis, we propose the heuristic $H_{\text{inverse}}$ that recommends which constraint formulation to choose when the constraints involve the inverse function application:

- **Integrated model $F_d1 + F_d2$:** If we have a capacity constraint, then use the formulation on $F_d1$ because we then get a global constraint. Otherwise, use the formulation on $F_d2$ because the constraint formulation in $F_d2$ avoids introducing extra Boolean variables and constraints.

- **Integrated model $F_d1 + S$:** If we have a capacity constraint, then in both $F_d1$ and $S$ we get a global constraint. In such a case, the heuristic cannot distinguish between the two formulations. Otherwise, if the constraint is not a capacity constraint, then the heuristic suggests using the formulation on $S$ because the constraint formulation in $S$ avoids introducing extra Boolean variables and constraints.

- **Integrated model $F_d2 + S$:** If we have a capacity constraint or a weighted capacity constraint, then use the constraint formulation in $S$ because we then get a global constraint. Otherwise, the heuristic cannot distinguish between the two formulations.

- **Hybrid model $F_d1 + F_d2$:** If we have a capacity constraint, then the formulation in $F_d2$ leads to a linear constraint and the formulation in $F_d1$ leads to a global constraint. In such a case, the heuristic cannot distinguish between the two formulations. Otherwise the heuristic suggests using the formulation in $F_d2$ because it avoids introducing extra Boolean variables and constraints.

- **Hybrid model $F_d2 + S$:** If we have a (weighted) capacity constraint, then the formulation in $F_d2$ leads to a linear constraint and the formulation in $S$ leads to a global constraint. In such a case, the heuristic cannot distinguish between the two formulations. Otherwise use the formulation in $F_d2$ when the resulting constraints in $F_d2$ are linear and the formulation in $S$ when the resulting constraints in $F_d2$ are non-linear.

Note that when the heuristic $H_{\text{inverse}}$ cannot distinguish between the two constraint formulations, we generate all the possible combinations. Thus, we will have three different integrated/hybrid models, where in the first one we use one of the constraint formulations, in the second we use the other constraint formulation, and in the third we use both constraint formulations.

**Set or arithmetic constraints involving the range set.** In Chapter 4, the range of the function is used in the expression of the objective function of the three problems. In
the optimization version of the GCP (Section 4.2) we have:

\[ \text{range}(\text{COLORING}) \]

and in the WLP (Section 4.3), we have:

\[ \sum_{(i,j) \in \text{SUPPLIER}} \text{supplycost}(i,j) + |\text{range}(	ext{SUPPLIER})| \times \text{maintcost} \]

Finally, in the BACDP (Section 4.4) we use:

\[ \text{max}(|\text{range}(	ext{LOAD})|) \]

Assume function \( F \) is declared as \( F: V \rightarrow W \). The following items summarize the number of extra variables introduced in \( Fd_1 \), \( Fd_2 \), and \( S \), their types and domains, and the number of linking constraints and their arity:

- In \( Fd_1 \), we introduce \(|W|\) Boolean variables, and \(|V| \times |W|\) binary linking constraints of the form \( Fd_1[i] = j \rightarrow \text{range}_F[j] = 1 \).
- In \( Fd_2 \), we introduce \(|W|\) Boolean variables, and \(|V| \times |W|\) linear binary linking constraints of the form \( Fd_2[i, j] \leq \text{range}_F[j] \).
- In \( S \), we introduce a set variable \( \text{range}_F \) that is a subset of \( W \), and \(|W|\) linking constraints of the form \( |S[j]| \geq 1 \rightarrow j \in \text{range}_F \). These linking constraints use a global cardinality constraint, but are fewer than the ones used in the representations \( Fd_1 \) and \( Fd_2 \).

Unfortunately, it is quite difficult to see which linking constraints are better even though they differ in the number and type of variables used as well the number of the linking constraints. So, let us check the resulting formulation of constraints involving the range set in the three representations.

In general, the constraints involving the range set would be the same for both representations \( Fd_1 \) and \( Fd_2 \) because they both represent the range set in the same way by using Boolean variables. However, when the constraint on the range set involves either the expression \( \text{max}(|\text{range}(F)|) \) or \( \text{min}(|\text{range}(F)|) \), the \text{max} and \text{min} global constraints (in Section 2.3) are used in \( Fd_1 \). Also, for some cases, set global constraints may be used in the \( S \) representation.

Based on this analysis, we propose the heuristic \( H_{\text{range}} \) for selecting a constraint formulation when some constraints involve the range set:

- Integrated model \( Fd_1 + Fd_2 \): If the constraint involves a \( \text{max}(|\text{range}(F)|) \) or \( \text{min}(|\text{range}(F)|) \) expression, then use the formulation on \( Fd_1 \) because we then get a global constraint. Otherwise, use either of the formulations because they are the same.
- Integrated model \( Fd_1 + S \): If the constraint involves a \( \text{max}(|\text{range}(F)|) \) or \( \text{min}(|\text{range}(F)|) \) expression, then use the formulation on \( Fd_1 \) because we then get a global constraint. Use the formulation in \( S \) whenever it results in a global constraint. Otherwise use the formulation in \( Fd_1 \) because the linking constraints only involve integer variables.
- Integrated model \( Fd_2 + S \): Use the formulation in \( S \) whenever it results in a global constraint, otherwise use the formulation in \( Fd_2 \) because the linking constraints only involve Boolean variables.
• Hybrid model $Fd_1 + Fd_2$: Use the formulation in $Fd_1$ when the resulting constraints are non-linear, otherwise use the formulation in $Fd_2$.

• Hybrid model $Fd_2 + S$: Use the formulation in $S$ when the resulting constraints are non-linear, otherwise use the formulation in $Fd_2$.

**Constraints involving the membership predicate.** The membership predicate on functions can be used to state some of the constraints. The predicate $(i, j) \in F$ may appear in a logical formula. In such a case, in all the three representations we obtain a binary constraint. We may as well have an arithmetic aggregate expression:

$$\sum_{(i,j) \in F} Q(i, j)$$

where $Q(i, j)$ is any arithmetic expression. In such a case, we need to introduce extra Boolean variables and constraints only in $S$. We also end up with a summation expression of $|V|$ terms in $Fd_1$, and $|V| \cdot |W|$ terms in $Fd_2$ and $S$. But the integer variables in $Fd_1$ have domains of size $|W|$ while we have Boolean variables in $Fd_2$ and $S$.

When we have a constraint of the form:

$$\forall (i, j) \in F \cdot P(i, j)$$

where $P(i, j)$ is any formula, no extra variables and constraints are needed in any representation. In this case, it is difficult to compare all three formulations.

Based on this analysis, we propose the heuristic $H_{membership}$ for selecting a constraint formulation when some constraints involve the membership predicate.

• Integrated model $Fd_1 + Fd_2$: The heuristic does not distinguish between the two formulations.

• Integrated model $Fd_1 + S$: When the constraint involves a summation expression, use the formulation in $Fd_1$ because it avoids introducing extra variables and constraints. Otherwise, the heuristic cannot distinguish between the two formulations.

• Integrated model $Fd_2 + S$: When the constraint involves a summation expression, use the formulation in $Fd_2$ because it avoids introducing extra variables and constraints. Otherwise, the heuristic cannot distinguish between the two formulations.

• Hybrid model $Fd_1 + Fd_2$: Use the formulation in $Fd_1$ when the resulting constraints are non-linear, otherwise use the formulation in $Fd_2$.

• Hybrid model $Fd_2 + S$: Use the formulation in $S$ when the resulting constraints are non-linear, otherwise use the formulation in $Fd_2$.

Note that when the heuristic $H_{membership}$ cannot distinguish between the two constraint formulations, we generate all the possible combinations.
Channelling constraints. Channelling constraints play a central role when two different models of the same problem are integrated because they provide the channel through which changes in the variables of one model are triggered into the variables of the other model. Despite the increase in the number of variables and constraints, domain pruning carried out in each model may be significantly improved, giving a more powerful model than any of the participating models.

In our case, we have three different ways of representing function variables. Given a total function variable $F: V \rightarrow W$, the first representation uses a 1d array $Fd_1$ indexed by $V$ and ranging over $W$, with the following meaning:

$$
\forall i \in V \cdot \forall j \in W \cdot Fd_1[i] = j \iff \langle i, j \rangle \in F
$$

In the second representation, we use a 2d 0/1 array $Fd_2$ indexed by $V$ and $W$, with the following meaning:

$$
\forall i \in V \cdot \forall j \in W \cdot Fd_2[i, j] = 1 \iff \langle i, j \rangle \in F
$$

Finally, in the third representation, we use an array $S$, indexed by $W$, of set variables that are subsets of $V$, with the following meaning:

$$
\forall i \in V \cdot \forall j \in W \cdot i \in S[j] \iff \langle i, j \rangle \in F
$$

Therefore, integrating $Fd_1$ and $Fd_2$ can be achieved if we add the following channelling constraints:

$$
\forall i \in V \cdot \forall j \in W \cdot Fd_1[i] = j \leftrightarrow Fd_2[i, j] = 1
$$

Integrating $Fd_1$ and $S$ can be achieved through the following channelling constraints:

$$
\forall i \in V \cdot \forall j \in W \cdot Fd_1[i] = j \leftrightarrow i \in S[j]
$$

Finally, integrating $Fd_2$ and $S$ can be achieved through the channelling constraints:

$$
\forall i \in V \cdot \forall j \in W \cdot Fd_2[i, j] = 1 \leftrightarrow i \in S[j]
$$

5.3 Implied constraints

ICs are logical consequences of the problem’s constraints. ICs may tighten the problem but do not remove solutions. The addition of ICs to the initial model may lead to a significant reduction in the search.

For certain functions, there are properties that must be satisfied. Given a function $F$ from $V$ into $W$, the following properties hold:

1. $F$ is a total function: $|V| \geq |\text{range}(F)|$
2. $F$ is a total injection: $|V| = |\text{range}(F)|$
3. $F$ is a total surjection: $|V| \geq |W|$
4. $F$ is a total bijection: $|V| = |W|$
5. $F$ is a partial function: $|\text{domain}(F)| \geq |\text{range}(F)|$
6. $F$ is a partial injection: $|\text{domain}(F)| = |\text{range}(F)|$
7. $F$ is a partial surjection: $|\text{domain}(F)| \geq |W|$

8. $F$ is a partial bijection: $|\text{domain}(F)| = |W|$

Except for the third and the fourth item in the list, each property can be added as an extra constraint to a given function model. Adding the appropriate property to the initial model written in $\mathcal{F}$ will result in a set of ICs in the corresponding generated $\mathcal{L}$ models. In particular, we will show in Chapter 6 that for the case of injective total functions, the generated ICs from the property $|V| = |\text{range}(F)|$ help pruning further inconsistent values that otherwise would not be pruned.

### 5.4 Summary

In this chapter, we showed how in practice function, sequence, and permutation variables can be represented in different ways in $\mathcal{L}$. We also showed the building blocks of how to mechanically map constraints on these variables into constraints in $\mathcal{L}$ depending on the representation choice. However, our approach does not cover arbitrarily nested expressions. We then explored the possibility of integrating different representations of functions, we analyzed different constraint formulations in a general way, and proposed the heuristics $\mathcal{H}_{\text{arithmetic}}$, $\mathcal{H}_{\text{inverse}}$, $\mathcal{H}_{\text{range}}$, and $\mathcal{H}_{\text{membership}}$ for selecting a constraint formulation for the combined models. Finally, we exploited properties that certain functions have in order to add a set of useful ICs to the generated $\mathcal{L}$ model from an $\mathcal{F}$ model.

The main contributions of this chapter may be summarized as follows:

- Function, sequence, and permutation variables and constraints on these variables can be mapped into elements of current CP languages producing, in general, more than one alternative.
- The introduction of function variables simplifies the automatic generation of alternate models.
- Many practical modelling tricks are encapsulated such as:
  - the choice of the variables and their domains;
  - the introduction of appropriate auxiliary and dual variables;
  - the introduction of channelling and auxiliary constraints;
  - the integration of models based on different representations of function variables;
  - the development of the heuristics $\mathcal{H}_{\text{arithmetic}}$, $\mathcal{H}_{\text{inverse}}$, $\mathcal{H}_{\text{range}}$, and $\mathcal{H}_{\text{membership}}$ that choose between different constraint formulations;
  - the addition of ICs.

The process of transforming models written in $\mathcal{F}$ generates, in general, more than one model in $\mathcal{L}$. The question now is which model to select? This is a very difficult and challenging task, which will be the concern of the next chapter.
Chapter 6
Model Selection

In Chapter 5, we showed how alternate $L$ models may be generated from a given function model. In this chapter, we address the question of model selection. Ideally, one should be able to partially order a set of models depending on the efficiency by which they would solve some problem instances. However, in general, this is a very challenging task indeed. There are many control parameters. First we have a set of models that vary on the set of variables used, their types and domain sizes, on the set of constraints imposed and their number, etc. Second, we have solution methods, which vary from CP solvers to ILP solvers and even different (CP) solvers have different characteristics. For instance, some CP solvers employ different search strategies while some enforce different levels of consistency on different constraints. Finally, we have problem instances, which have different characteristics such as the input size and the constraint tightness.

In this chapter, we will first argue why it is difficult to solve the general problem of model selection in Section 6.1. Second, in Section 6.2, we propose possible alternate $L$ models of total injection problems. Third, in Section 6.3, we theoretically compare the proposed $L$ models of injection problems by using the measure of constraint tightness parameterized by the level of consistency being enforced [81]. We also make an asymptotic comparison of the time required to enforce different consistency levels and run some experiments on three injection problems. Finally, we summarize our contributions of this chapter in Section 6.7.

6.1 Difficulty of model selection

The choice of a good model may result in a huge reduction in the search cost. The choice of a bad model may result in huge increase in the search cost. Why are some models good while others are bad? Why is it considered an art to come up with a good model?

Given a CSP $P$, the task of mapping the problem $P$ into a constraint program $C$ can be viewed as a composition of modelling decisions summarized as follows:

- **The choice of the variables:** we can choose between integer variables versus set variables. The number and domains of such variables may vary.

- **The choice of the constraint formulation:** different formulations have different properties. Constraints may be binary, linear, or global as well as they may vary in number.
• The choice of the labelling strategy: developing a labelling strategy other than the
default one is optional, but often a necessary step to enhance the performance.
However, developing a good labelling strategy is still an art.

• The choice of implied constraints: the addition of implied constraints is optional,
but may help reduce the search cost. Implied constraints are logical consequences
of a set of constraints, hence their number may be infinite.

• The choice of symmetry-breaking constraints: when we have symmetry in a model,
it may be beneficial to add symmetry-breaking constraints so as to avoid wasting
time in exploring symmetric parts of the search space.

• The choice of the solution method: we can either pick an off-the-shelf black-box
solver or develop a problem-specific solver. Different solvers may vary in the search
algorithm used as well as in the level of consistency being enforced.

Each of these modelling decisions requires a non-deterministic choice among a very large
(and possibly infinite) number of possibilities. Therefore, the space of possible models is
very large (and possibly infinite). It is clear that trying to find the best possible model
cannot be achieved with certainty. Thus, it is just reasonable to restrict the size of the
space of models. In other words, and as proposed in [5], a model selection method can be
defined as finding the model among a given set such that it has the minimum cost, where
cost is the required time to solve a particular problem. Furthermore, this method must
satisfy the following [5]:

• Generality: the method has to be general so that it can be applied to a wider range
of problems.

• Accuracy: ideally, the method should be accurate in choosing the most effective
model.

• Low cost: the method should have a low cost in terms of time.

However, it is unlikely that a method satisfying all these conditions exists. One might
think of a brute force approach where all the models are solved on a particular instance
and then the most effective model is chosen. Even though this brute force method is
general and accurate, it has a very high cost. Thus, approximate cost functions that can
be tested quickly and provide a good estimate on the actual cost of solving a particular
model should be developed and adopted instead.

We also think that even with a restricted set of models, the task of selecting the most
effective model is still very difficult. Some of the reasons that contribute to the difficulty
of the task are:

1. Sensitivity of solvers to instance distributions: the performance of a particular solver
on a particular model is instance-dependent. For a given problem a solver can
perform well for some (distribution on the) instances, but very poorly on others
(see [73, 54]).

2. Many control parameters: there are many parameters that characterize a model
such as the number of variables, their domain sizes, the number of constraints and
their arity, the constraint tightness, the topology of the constraint graph, the level of
consistency being enforced, etc. Deciding which parameters could tell us about the required solving time is a little understood art. Thus, developing good approximate cost functions is tough.

3. **Conflict of concerns**: while some models employ less variables but poor symmetry-breaking constraints (if any), others may be better suited for breaking more symmetry, but with more variables and extra constraints. It may be the case that different formulations have different strengths. Deciding which factor is more important is not obvious. It may be the case that breaking more symmetry is more beneficial when the problem has a large amount of symmetry, or it may as well be more beneficial to have less variables and less constraints for some problem where symmetry is not the real bottleneck.

Despite the difficulty of model selection, there are some approaches that are promising. For instance, Borrett and Tsang developed a framework for systematic model selection. They demonstrate their approach on the example of assessing the addition of a certain class of ICs to an original model [5]. The evaluation heuristic used is based on an extension of the theoretical complexity estimates proposed by Nadel [57]. Their experimental results show that the approach is very promising. However, with this approach one needs the problem instance to be an explicit input to the methods. Thus, the methods need to have a low cost, as is the case in [5]. Taking a different approach, Walsh introduces the measure of constraint tightness parameterized by the level of local consistency being enforced [81]. This measure works for any variable domains, and thus does not require the problem instance as an input. Walsh used this measure to compare different models of permutation problems and shows very interesting results [81].

6.2 Models of injection problems

We demonstrate our approach to model selection by focusing on models of injection problems. The choice of injection problems is due to the fact that many problems can be modelled as injection problems. Our restriction to only injection problems is due to space and time limitations, but is enough to show significant steps towards model selection.

There are many problems that can be modelled as injection problems. These problems can be scheduling problems, combinatorial graph problems, crypt-arithmetic puzzles, etc. For instance, the tournament scheduling problem (modelled in [79] for even number of teams) with an odd number of teams amounts to finding, for each week, an injection from the set \( \text{periods} \times \text{slots} \) into the set of \( \text{teams} \). To model the graceful graph problem [60], an injection from the set of vertices into the set of labels, as well as a permutation of the set of edges ought to be found. Finally, solving the send-more-money puzzle amounts to finding an injection from the set of characters into the set of digits.

Injection problems can be modelled as constraint satisfaction problems. The straightforward formulation would be to have as many variables as the elements of the target set that range over the source set, which captures a total function. To enforce that the function is injective, we would need to state an alldifferent constraint among all the variables (alternative \( b \) of Figure 5.3).

Given an injection problem, in what follows we present some possible models. We present two primal models, three dual models, three minimal integrated models that just link the primal and dual variables through channelling constraints and have no extra
Variables: \( F_1, \ldots, F_n \in C \)

Constraints: all\( \text{different} \{(F_1, \ldots, F_n)\} \)

Figure 6.1: The \( \forall \) model

Variables: \( F_1, \ldots, F_n \in C \)

Constraints: \( \forall i, j \in D: i \neq j. F_i \neq F_j \)

Figure 6.2: The \( \neq \) model

Primal models

An injection \( F: D \longrightarrow C \) is a function where \( D = \{1, \ldots, n\}, C = \{1, \ldots, m\}, n \leq m \); and injective(\( F \)) is a bijection when \( n = m \). Walsh have compared different models of bijection problems [81]. We here do not consider the possible models for injections that result by transforming the injection into a bijection by adding extra values to \( D \) and declaring \( F \) now as:

\[
F: D \cup \{n + 1, \ldots, m\} \longrightarrow C
\]

and enforcing bijective(\( F \)) because these models are already discussed in [81]. We here only consider injection problems where \( n < m \).

We start by presenting two primal models of an injection problem, which are already given in Chapter 5:

- The primal all\( \text{different} \) model (denoted by \( \forall \)): We introduce \( n \) integer variables \( F_1, \ldots, F_n \) such that every \( F_i \in C \), and state an all\( \text{different} \) constraint among them. This model is shown in Figure 6.1 (this corresponds to alternative \( a \) of Figure 5.3).

- The primal not-equals model (denoted by \( \neq \)): We introduce \( n \) integer variables, \( F_1, \ldots, F_n \) such that every \( F_i \in C \), and instead of stating the global constraint all\( \text{different} \), we state binary not-equals constraints between every two distinct variables. This model is shown in Figure 6.2 (this corresponds to alternative \( b \) of Figure 5.3).

Dual models

The inverse function of an injection is a partial function. Modelling the inverse function can be achieved by viewing the values in the previous models as variables and vice versa. Exploiting the dual viewpoint on injection problems leads to the following dual models.

- The dual all\( \text{different} \) model (denoted by \( \forall_d \)): We introduce \( m \) (dual) integer variables, \( G_1, \ldots, G_m \) such that every \( G_i \in D \cup \{n + 1, \ldots, m\} \), where values \( n + 1 \) through \( m \) are extra values introduced to be able to state an all\( \text{different} \) constraint among the dual variables as some of the elements of the target set may not be the image of any element in the source set. We then state an all\( \text{different} \) constraint among all dual variables. This model is shown in Figure 6.3.
Variables: $G_1, \ldots, G_m \in D \cup \{n + 1, \ldots, m\}$
Constraints: $\textit{alldifferent}([G_1, \ldots, G_m])$

Figure 6.3: The $\forall_d$ model

Variables: $G_1, \ldots, G_m \in D \cup \{n + 1, \ldots, m\}$
Constraints: $\forall i, j \in C: i \neq j \cdot G_i \neq G_j$

Figure 6.4: The $\neq_d$ model

- The dual not-equals model (denoted by $\neq_d$): We introduce $m$ (dual) integer variables, $G_1, \ldots, G_m$ such that every $G_i \in D \cup \{n + 1, \ldots, m\}$, and state a not-equals constraint between every two distinct variables. This model is shown in Figure 6.4.

- The dual occurs model (denoted by $o$): We introduce $m$ (dual) integer variables, $G_1, \ldots, G_m$ such that every $G_i \in D \cup \{n + 1\}$, where value $n + 1$ is a single extra value introduced so that the dual variables that are not images of any element in the range get mapped to it. We then state the global constraint $\textit{occurs}(i, [G_1, \ldots, G_m], 1)$, for every $i \in D$, to enforce that every value $i$ in the source set gets assigned to exactly one variable $G_k$. This model is shown in Figure 6.5.

Minimal integrated models

The primal and dual variables can be linked through channelling constraints resulting in minimal integrated models. We here present three possible ways of minimal integration of primal and dual variables.

- The first minimal integrated model (denoted by $c_1$): We introduce $n$ primal integer variables, $F_1, \ldots, F_n$ such that every $F_i \in C$, and $m$ dual integer variables, $G_1, \ldots, G_m$ such that every $G_i \in D$. We then just need to link the primal and dual variables through the following channelling constraints:

$$\forall i \in D \cdot \forall j \in C \cdot F_i = j \rightarrow G_j = i$$

These channelling constraints alone define the desired injection. This model is shown in Figure 6.6 (this corresponds to alternative $c$ of Figure 5.3).

- The second minimal integrated model (denoted by $c_2$): We introduce $n$ primal integer variables, $F_1, \ldots, F_n$ such that every $F_i \in C$, and $m$ dual integer variables, $G_1, \ldots, G_m$ such that every $G_i \in D \cup \{n + 1, \ldots, m\}$. We then need just to link the primal and dual variables through the following channelling constraints:

$$\forall i \in D \cdot \forall j \in C \cdot F_i = j \leftrightarrow G_j = i$$

These channelling constraints alone define the desired injection. Note that channelling constraints of the form $F_i = j \rightarrow G_j = i$ would be sufficient. However, we add the other side of the implication ($G_j = i \rightarrow F_i = j$) because it allows the domain pruning on the dual variable to be triggered to the primal variables. This cannot be done for the channelling constraints in model $c_1$. This model is shown in Figure 6.7.
Proof:
For the model $D$ have domain $\{n + 1\}$. Now, consider a solution to model $c_1$. The primal variables alone define an injection from $D$ into $C$. We just need to show that every two distinct primal variables get mapped to distinct images. This is a contradiction as $i \neq j$. Since we have the channelling constraint $F_i = k \rightarrow G_k = i$, $G_k$ must be $i$. We also have the channelling constraint $F_j = k \rightarrow G_k = j$, thus $G_k$ must also be $j$. Hence, $i = j$. This is a contradiction as $i \neq j$, so all the primal variables must have different values. Therefore, the model $c_1$ defines an injection from $D$ into $C$.

For the model $c_2$, the primal variables have domain $D$ while the dual variables have domain $D \cup \{n + 1\}$. Now, consider a solution to model $c_2$. The primal variables alone define a total function from $D$ into $C$. We just need to show that every two distinct primal variables get mapped to distinct images. Since we have the channelling constraint $F_i = F_j = k$ when $i \neq j$. This is a contradiction as $i \neq j$. So all the primal variables must have different values. Therefore, the model $c_2$ defines an injection from $D$ into $C$.

For the model $c_3$, the primal variables have domain $C$ while the dual variables have domain $D \cup \{n + 1\}$. Now, consider a solution to model $c_3$. The primal variables alone define an injection from $D$ into $C$. We just need to show that every two distinct primal variables get mapped to distinct images. Since we have the channelling constraint $F_i = F_j = k$ when $i \neq j$. This is a contradiction as $i \neq j$. So all the primal variables must have different values. Therefore, the model $c_2$ defines an injection from $D$ into $C$.
Variables: \( F_1, \ldots, F_n \in C \)
\( G_1, \ldots, G_m \in D \cup \{ n + 1, \ldots, m \} \)

Constraints: \( \forall i \in D. \forall j \in C. F_i = j \iff G_j = i \)

Figure 6.7: The \( c_2 \) model

Variables: \( F_1, \ldots, F_n \in C \)
\( G_1, \ldots, G_m \in D \cup \{ n + 1 \} \)

Constraints: \( \forall i \in D. \forall j \in C. F_i = j \iff G_j = i \)

Figure 6.8: The \( c_3 \) model

define a total function from \( D \) into \( C \). We just need to show that every two distinct primal variables get mapped to distinct images. So, let’s assume \( F_i = F_j = k \) when \( i \neq j \). Since we have the channelling constraint \( F_i = k \iff G_k = i \), \( G_k \) must be \( i \). We also have the channelling constraint \( F_j = k \iff G_k = j \), thus \( G_k \) must also be \( j \). Hence, \( i = j \). This is a contradiction as \( i \neq j \). So all the primal variables must have different values. Therefore, the model \( c_3 \) defines an injection from \( D \) into \( C \). QED.

Models with implied constraints

Given an injection \( F : D \rightarrow C \), the property:

\[ |\text{range}(F)| = |D| \]

presented in Section 5.3 must also hold, and when added to the injection model will get translated into a set of ICs.

By applying the results in Chapter 5, \( \text{range}(F) \) is represented by the Booleans \( r_1, \ldots, r_m \) and the following linking constraints are added:

\[ \forall i \in D, j \in C : F_i = j \rightarrow r_j = 1 \]

when we link the primal variables \( F_i \) with the Boolean variables \( r_j \). Alternatively, we can link the dual variables \( G_j \) in \( c_2 \) and \( c_3 \) (but not in \( c_4 \)) to the Boolean variables \( r_j \) by using fewer linking constraints as follows:

\[ \forall j \in C : G_j \in D \leftrightarrow r_j = 1 \]

Enforcing \( |\text{range}(F)| = |D| \) is represented as:

\[ \left( \sum_{j \in C} r_j \right) = |D| \]

We will only consider the addition of the ICs to models \( c_2 \) and \( c_3 \) in our theoretical comparison because there are fewer linking constraints.

Whenever, the ICs and the linking constraints are introduced in \( c_2 \) and \( c_3 \), we will use \( |W| \) to denote a model that includes the ICs and the linking constraints. For instance, model \( c_2|W| \) denotes the model \( c_2 \) with the additional ICs and linking constraints and are shown in Figure 6.9 and Figure 6.10, respectively.
Integrated models

Other integrated models can be constructed by adding extra constraints, from either the primal or dual models or both, to the minimal integrated model. We will consider the following integrated models:

- Integrated model $\neq c_1$: where the not-equals constraints on the primal variables are added to minimal integrated model $c_1$.
- Integrated model $c_2 \neq d$: where the not-equals constraints on the dual variables are added to minimal integrated model $c_2$.
- Integrated model $\neq c_2 \neq d$: where the not-equals constraints on the primal variables and the not-equals constraints on the dual variables are added to minimal integrated model $c_2$.
- Integrated model $c_2[W]$: where the ICs are added to minimal integrated model $c_2$.
- Integrated model $\neq c_2[W]$: where the not-equals constraints on the primal variables and the ICs are added to minimal integrated model $c_2$.
- Integrated model $c_3\alpha$: where the occurs constraints on the dual variables are added to minimal integrated model $c_3$.
- Integrated model $\neq c_3\alpha$: where the not-equals constraints on the primal variables and the occurs constraints on the dual variables are added to minimal integrated model $c_3$.
- Integrated model $c_3[W]$: where the ICs are added to minimal integrated model $c_3$.
- Integrated model $\neq c_3[W]$: where the not-equals constraints on the primal variables and the ICs are added to minimal integrated model $c_3$.

There will, of course, typically be other constraints which depend on the nature of the injection problem. In the theoretical comparison, we do not consider the contribution of such additional constraints to pruning. The contribution of such constraints will be considered in the empirical study.

87
6.3 A theoretical comparison of models of injection problems

We here use the measure of constraint tightness parameterized by the level of local consistency being enforced [81] to compare different models of injection problems. In this section, the constraint tightness measure [81] is used to theoretically compare the different models with respect to local consistency properties, including arc-consistency, as well as to algorithms that either maintain a restricted form of arc-consistency (FC) or maintain (generalized) arc-consistency during search (MAC and MGAC), and bounds consistency. The theoretical comparison is enhanced by analyzing the asymptotic costs of maintaining different local consistency properties. Finally, some empirical results are presented in order to add details to the theoretical results by considering the contribution of additional constraints depending on the nature of the injection problem.

6.3.1 Constraint tightness

To compare how different models of injection problems prune the search tree, we use the definition of Walsh [81], which introduces a measure of constraint tightness. This definition of constraint tightness is strongly influenced by the way local consistency properties are compared [18]. Indeed, the definition is parameterized by a local consistency property since the amount of pruning provided by a set of constraints can depend on the level of local consistency being enforced. Walsh also argues that the measure of constraint tightness would also be useful in a number of other applications such as reasoning about the value of implied constraints.

In [81], the following definitions are presented:

- A set of constraints $A$ is $\Phi$-consistent iff every constraint in $A$ is $\Phi$-consistent.
- A set of constraints $A$ is as tight as a set $B$ with respect to $\Phi$-consistency (written $\Phi_A \Rightarrow \Phi_B$) iff, given any domains for their variables, if $A$ is $\Phi$-consistent then $B$ is also $\Phi$-consistent. Note that tightness is over all possible domains for the variables. It thus measures the possible pruning of domains during search as variables are instantiated and domains pruned (possibly by other constraints in the problem).
- A set of constraints $A$ is tighter than a set $B$ wrt $\Phi$-consistency (written $\Phi_A \rightarrow \Phi_B$) iff $\Phi_A \Rightarrow \Phi_B$ but not $\Phi_B \Rightarrow \Phi_A$.
- A set of constraints $A$ is incomparable to $B$ wrt $\Phi$-consistency (written $\Phi_A \otimes \Phi_B$) iff neither $\Phi_A \Rightarrow \Phi_B$ nor $\Phi_B \Rightarrow \Phi_A$.
- A set of constraints $A$ is equivalent to $B$ wrt $\Phi$-consistency (written $\Phi_A \leftrightarrow \Phi_B$) iff both $\Phi_A \Rightarrow \Phi_B$ and $\Phi_B \Rightarrow \Phi_A$.

Constraint tightness has some monotonicity and fixed-point properties [81]:

Theorem 2 (monotonicity and fixed-point for AC)

1. $AC_{A \cup B} \Rightarrow AC_A \Rightarrow AC_{A \cup B}$
2. $AC_A \Rightarrow AC_B$ implies $AC_{A \cup B} \leftrightarrow AC_A$

88
Proof: See [81]. QED.

Similar monotonicity and fixed-point results hold for BC, RPC, PIC, SAC, ACPC, and GAC [81]. Walsh also extends these definitions to compare constraint tightness wrt search algorithms like MAC and FC that maintain some local consistency during search:

- A set of constraints \( A \) is as tight as \( B \) wrt algorithm \( X \) (written \( X_A \leadsto X_B \)) iff, given any fixed variable and value ordering and any domains for their variables, \( X \) visits no more nodes on \( A \) than on \( B \).
- A set of constraints \( A \) is tighter than \( B \) wrt algorithm \( X \) (written \( X_A \rightarrow X_B \)) iff \( X_A \leadsto X_B \) but not \( X_B \leadsto X_A \).

Similar monotonicity and fixed-point results to the one for AC are given for FC, MAC and MGAC [81]. Finally, we write \( X_A \Rightarrow X_B \) if \( X_A \rightarrow X_B \) and there is a problem on which \( X \) visits exponentially fewer branches with \( A \) than \( B \) [81].

### 6.3.2 Arc-consistency

We first prove that, with respect to arc-consistency, the channelling constraints of \( c_1 \) are as tight as the primal not-equals constraints of \( \neq \), but less tight than the primal alldifferent constraint of \( \forall \). Therefore, the amount of pruning achieved in \( \forall \) will be at least as much as \( c_1 \) and \( \neq \). However, \( c_1 \) and \( \neq \) will achieve exactly the same amount of pruning.

Then, we prove that the channelling constraints of \( c_2 \) are as tight as the primal not-equals constraints of \( \neq \), but less tight than the channelling and dual not-equals constraints of \( c_2 \neq d, \) which are less tight than the channelling and ICs of \( c_2[\forall] \), which are less tight than the primal alldifferent constraint of \( \forall \). This shows that we achieve more pruning in \( c_2 \) than in \( \neq \) only when either the dual not-equals constraint or the ICs are added to \( c_2 \), but still less pruning than in \( \forall \). Furthermore, the ICs constraints bring more pruning to \( c_2 \) than the dual not-equals constraints.

Finally, we prove that the channelling constraints of \( c_3 \) are as tight as the primal not-equals constraints of \( \neq \) and the channelling and dual occurs constraints of \( c_3[\forall] \), but less tight than the channelling and ICs of \( c_3[\forall] \), which are less tight than the primal alldifferent constraint of \( \forall \). This shows that \( c_3 \) achieves the same amount of pruning as \( \neq \). Furthermore, adding the dual occurs constraints to the channelling constraints of \( c_3 \) does not bring any more pruning. Thus, the dual occurs constraints are redundant. However, the addition of the ICs to the channelling constraints of \( c_3 \) allows more pruning, but still not as much as the pruning achieved in \( \forall \).

In the following we use the notation \( F_1 = F_2 = \ldots = F_n = D \) to stand for \( F_i \in D \land F_2 \in D \land \ldots \land F_n \in D \).

**Theorem 3** On an injection problem:

\[
\text{GAC}_\forall \rightarrow \text{AC}_{\neq c_1} \leftrightarrow \text{AC}_{c_1} \leftrightarrow \text{AC}_{\neq}
\]

**Proof:** To show \( \text{GAC}_\forall \rightarrow \text{AC}_{c_1} \), consider an injection problem whose primal alldifferent constraint is GAC. Suppose the channelling constraint between \( F_i \) and \( G_j \) was not AC. Then \( F_i \) is set to \( j \) and \( G_j \) has \( i \) eliminated from its domain. But this is not possible by the construction of the primal and dual model. Hence the channelling constraints are all AC. To show strictness, consider an injection problem in which \( F_1 = F_2 = F_3 = \{1, 2\} \) and \( G_1 = G_2 = G_3 = G_4 = \{1, 2, 3\} \). This is \( \text{AC}_{c_1} \) but not \( \text{GAC}_\forall \).

89
To show $\text{AC}_c \leftrightarrow \text{AC}_d$, suppose that the channelling constraints are AC. Consider a not-equals constraint, $F_i \neq F_j$ (where $i \neq j$) that is not AC. Now, $F_i$ and $F_j$ must have the same singleton domain, $\{k\}$. Consider the channelling constraint between $F_i$ and $G_k$. The only AC value for $G_k$ is $i$. Similarly, the only AC value for $G_k$ in the channelling constraint between $F_j$ and $G_k$ is $j$. But $i \neq j$. Hence, $G_k$ has no AC values. This is a contradiction as the channelling constraints are AC. Hence all not-equals constraints are AC. Now suppose that the not-equals constraints are AC. Consider a channelling constraint between $F_i$ and $G_j$ that is not AC. Then $F_i$ is set to $j$ and $G_j$ has $i$ eliminated from its domain. But for $i$ to be eliminated from the domain of $G_j$, some other primal variable, say $F_k$ where $k \neq i$, is set to $j$, which eliminate $j$ from the domain of $F_i$ (since the not-equals constraints are AC). Hence, it is not possible to set $F_i$ to $j$ and $G_j$ has $i$ eliminated from its domain. Thus, all channelling constraints are AC.

To show $\text{AC}_{\neq c} \leftrightarrow \text{AC}_{\neq d}$, by monotonicity, we have $\text{AC}_{\neq c} \rightarrow \text{AC}_{\neq d}$. To show the reverse, consider an injection problem whose ICs are not GAC. Then there exists at least one not-equals constraint that is not AC. Suppose that the not-equals constraint between $F_i$ and $F_j$ where $i \neq j$ is not AC. Then $F_i$ and $F_j$ must have the same singleton domain, say $\{k\}$. Hence the domain of the dual variable $G_k$ includes $i$ and $j$. Then, the channelling constraints are not AC. This is a contradiction. QED.

**Theorem 4** On an injection problem:

$$\text{GAC}_c \rightarrow \text{GAC}_{c|\{W\}} \rightarrow \text{AC}_{\neq c|\{W\}} \rightarrow \text{AC}_{\neq c} \rightarrow \text{AC}_c \rightarrow \text{AC}_c$$

**Proof:** To show $\text{GAC}_c \rightarrow \text{GAC}_{c|\{W\}}$, consider an injection problem whose primal _alldifferent_ constraint is GAC. Now, suppose the ICs are not GAC. There exist at least $m - n + 1$ dual variables with domains not in $D$. Because of the channelling constraints, $m - n + 1$ primal variables should have $n$ values pruned from their domains, thus $m - n + 1$ primal variables have the same domain size, which is $m - n$. But, since the _alldifferent_ constraint is GAC, this is a contradiction. To show strictness, consider an injection in which $F_1 = F_2 = F_3 = \{1,2\}$, $F_4 = \{1,2,3,4,5,6\}$, $G_1 = G_2 = \{1,2,3,4,5,6\}$, and $G_3 = G_4 = G_5 = G_6 = \{4,5,6\}$. This is $\text{GAC}_{c|\{W\}}$, but not $\text{GAC}_c$.

To show $\text{GAC}_{\neq d|\{W\}} \rightarrow \text{AC}_{\neq d}$, consider an injection problem whose $\text{GAC}_{\neq d|\{W\}}$. Suppose the not-equal constraint between $G_i$ and $G_j$ was not AC. Then, in the first case, $G_i = G_j = k$ and $k < n + 1$, which is impossible because the channelling constraints $F_k = i \leftrightarrow G_i = k$ and $F_k = j \leftrightarrow G_j = k$ are AC. In the second case, $k$ would be greater than $n$, which is impossible by construction of the primal and dual model. Hence all binary not-equal constraints on the dual variables are AC. To show strictness, consider an injection in which $F_1 = F_2 = F_3 = \{1,2\}$, $G_1 = G_2 = \{1,2,3,4,5\}$, and $G_3 = G_4 = G_5 = \{4,5\}$. This is $\text{AC}_{\neq d}$ but not $\text{GAC}_{\neq d|\{W\}}$.

To show $\text{GAC}_c \rightarrow \text{AC}_{\neq d}$, consider an injection problem whose primal _alldifferent_ constraint is GAC. Suppose the channelling constraint between $F_i$ and $G_j$ was not AC. Then, either $F_i$ is set to $j$ and $G_j$ has $i$ eliminated from its domain, or $G_j$ is set to $i$ (where $i \leq n$) and $F_i$ has $j$ eliminated from its domain. But this is not possible by the construction of the primal and dual model. Hence the channelling constraints are all AC. Now, suppose that a not-equals constraint between $G_i$ and $G_j$ is not AC. Then $G_i$ and $G_j$ must have the singleton domain, say $\{k\}$. When $k \leq n$, and because of the channelling constraints between $F_k$ and $G_k$ and between $F_k$ and $G_j$, we must have $F_k = i = j$, which is impossible since the primal _alldifferent_ constraint is GAC. The case when $k > n$ is
impossible by the construction of the primal and dual model. To show strictness, consider an injection problem in which \( F_1 = F_2 = F_3 = \{1, 2\} \), and \( G_1 = G_2 = \{1, 2, 3, 4, 5\} \), and \( G_3 = G_4 = G_5 = \{4, 5\} \). This is AC\(_{\forall \neq \phi}\) but not GAC\(_{\forall}\).

To show AC\(_{\forall \neq \phi}\) \(\rightarrow\) AC\(_{\forall}\), by monotonicity, we have AC\(_{\forall \neq \phi}\) \(\rightarrow\) AC\(_{\forall}\). To show strictness, consider an injection problem in which \( F_1 = F_2 = F_3 = \{1, 2\} \), and \( G_1 = G_2 = \{1, 2, 3, 4\} \), and \( G_3 = G_4 = \{4\} \). This is AC\(_{\forall}\) but not GAC\(_{\forall \neq \phi}\).

To show AC\(_{\forall}\) \(\leftrightarrow\) AC\(_{\phi}\), suppose that the channelling constraints are AC. Consider a not-equals constraint, \( F_i \neq F_j \) (where \( i \neq j \)) that is not AC. Now, \( F_i \) and \( F_j \) must have the same singleton domain, \( \{k\} \). Consider the channelling constraint between \( F_i \) and \( F_k \). The only AC value for \( G_k \) is \( i \). Similarly, the only AC value for \( G_k \) in the channelling constraint between \( F_j \) and \( G_k \) is \( j \). But \( i \neq j \). Hence \( G_k \) has no AC values. This is a contradiction as the channelling constraints are AC. Hence all not-equals constraints are AC. To show the reverse, suppose that the not-equals constraints are AC. Consider a channelling constraint, \( F_i = j \leftrightarrow G_j = i \), that is not AC. Then, either \( F_i \) is set to \( j \) and \( G_j \) has \( i \) eliminated from its domain, or \( G_j \) is set to \( i \) and \( F_i \) has \( j \) eliminated from its domain. But, for \( i \) to be eliminated from the domain of \( G_j \), some other primal variable, say \( F_k \), where \( k \neq i \), is set to \( j \), which will eliminate \( j \) from the domain of \( F_i \) (since the not-equals constraints are AC). Hence it is not possible to set \( F_i \) to \( j \) and \( G_j \) has \( i \) eliminated from its domain. For \( G_j \) to be set to \( i \), all the other values must be removed from its domain, but there is no way to remove any of the values bigger than \( n \) from the domain of \( G_j \), because at most we have \( n \) primal variables. Thus, all channelling constraints are AC.

To show AC\(_{\forall \neq \phi}\) \(\rightarrow\) AC\(_{\forall \neq \phi}\), by monotonicity, we have AC\(_{\forall \neq \phi}\) \(\rightarrow\) AC\(_{\forall \neq \phi}\). To show the reverse, consider an injection problem which is AC\(_{\forall \neq \phi}\) but not AC\(_{\forall \neq \phi}\). Then there exists at least one not-equals constraint on the primal variables, say \( F_i \) and \( F_j \), that is not AC. So, \( F_i = F_j = k \) and \( i \neq j \). But, since the channelling constraints are AC, we have \( G_k = i \) and \( G_k = j \). Hence, \( i = j \). This is a contradiction as \( i \neq j \). QED.

**Theorem 5** On an injection problem:

\[
\text{GAC}_\forall \rightarrow \text{GAC}_{\forall|W|} \rightarrow \text{GAC}_{\forall \neq \phi} \leftrightarrow \text{GAC}_{\phi} \leftrightarrow \text{AC}_{\forall} \leftrightarrow \text{AC}_\phi.
\]

**Proof:**

To show GAC\(_{\forall}\) \(\rightarrow\) GAC\(_{\forall|W|}\), consider an injection problem whose primal alldifferent constraint is GAC. Now, suppose the ICs are not GAC. There exists, at least, \( m - n + 1 \) dual variables with domain \( \{n + 1\} \). Because of the channelling constraints, \( m - n + 1 \) primal variables should have \( n \) values pruned from their domains, thus \( m - n + 1 \) primal variables have the same domain size, which is \( m - n \). But, since the alldifferent constraint is GAC, this is a contradiction. To show strictness, consider an injection in which \( F_1 = F_2 = F_3 = \{1, 2\} \), \( F_4 = \{1, 2, 3, 4, 5\} \), \( G_1 = G_2 = \{1, 2, 3, 4, 5\} \), and \( G_3 = G_4 = G_5 = \{4, 5\} \). This is GAC\(_{\forall|W|}\), but not GAC\(_{\forall}\).

To show GAC\(_{\forall}\) \(\rightarrow\) AC\(_{\forall}\), consider an injection problem whose primal alldifferent constraint is GAC. Suppose the channelling constraint between \( F_i \) and \( G_j \) was not AC. Then, either \( F_i \) is set to \( j \) and \( G_j \) has \( i \) eliminated from its domain, or \( G_j \) is set to \( i \) (where \( i \leq n \)) and \( F_i \) has \( j \) eliminated from its domain. But this is not possible by the construction of the primal and dual model. Hence the channelling constraints are all AC. Now, suppose that an occurs constraint that is not GAC. Hence at least two variables, say \( G_i \) and \( G_j \), have a singleton domain, say \( \{k\} \). When \( k \leq n \), and because of the channelling constraints between \( F_k \) and \( G_i \) and between \( F_k \) and \( G_j \), we must have \( F_k = i = j \), which
is impossible since the primal \textit{alldifferent} constraint is GAC. The case when \( k > n \) is impossible by the construction of the primal and dual model. To show strictness, consider an injection problem in which \( F_1 = F_2 = F_3 = \{1, 2\} \), and \( G_1 = G_2 = \{1, 2, 3, 4\} \), and \( G_3 = G_4 = \{4\} \). This is GAC\(_{\wedge}\), but not GAC\(_{\vee}\).

To show GAC\(_{\wedge}\) \( \not\rightarrow \) AC\(_{\wedge}\), by monotonicity, we have GAC\(_{\wedge}\) \( \not\sim \) AC\(_{\wedge}\). To show the reverse, suppose that all the channelling constraints are AC. Consider an occurs constraint that is not GAC. Then there exist (at least) two dual variables, say \( G_i \) and \( G_j \) with \( i \neq j \), that have the same domain \( \{k\} \) with \( k \leq n \). But since the channelling constraints are AC, we have \( F_k = i = j \). Hence, \( i = j \). This is a contradiction. So, all occurs constraints are GAC.

To show GAC\(_{\wedge}V\) \( \not\rightarrow \) GAC\(_{\wedge}\), by monotonicity, we have GAC\(_{\wedge}V\) \( \not\sim \) GAC\(_{\wedge}\). To show strictness, consider an injection in which \( F_1 = F_2 = F_3 = \{1, 2\} \), \( G_1 = G_2 = \{1, 2, 3, 4\} \), and \( G_3 = G_4 = \{4\} \). This is GAC\(_{\wedge}\) but not GAC\(_{\wedge}V\).

To show AC\(_{\wedge}\) \( \not\rightarrow \) AC\(_{\wedge}\), suppose that the channelling constraints are AC. Consider a not-equals constraint, \( F_1 \neq F_2 \) (where \( i \neq j \)) that is not AC. Now, \( F_1 \) and \( F_2 \) must have the same singleton domain, \( \{k\} \). Consider the channelling constraint between \( F_1 \) and \( G_k \). The only AC value for \( G_k \) is \( i \). Similarly, the only AC value for \( G_k \) in the channelling constraint between \( F_2 \) and \( G_k \) is \( j \). But \( i \neq j \). Hence \( G_k \) has no AC values. This is a contradiction as the channelling constraints are AC. Hence all not-equals constraints are AC. To show the reverse, suppose that the not-equals constraints are AC. Consider a channelling constraint, \( F_k = j \rightarrow G_j = i \), that is not AC. Then, either \( F_k \) is set to \( j \) and \( G_j \) has \( i \) eliminated from its domain, or \( F_k \) is set to \( i \) and \( F_j \) has \( j \) eliminated from its domain. But, for \( i \) to be eliminated from the domain of \( G_j \), some other primal variable, say \( F_k \), where \( k \neq i \), is set to \( j \), which will eliminate \( j \) from the domain of \( F_k \) (since the not-equals constraints are AC). Hence it is not possible to set \( F_k \) to \( j \) and \( G_j \) has \( i \) eliminated from its domain. For \( G_k \) to be set to \( i \), all the other values must be removed from its domain, but there is no way to remove any of the values bigger than \( n \) from the domain of \( G_j \), because we have at most \( n \) primal variables. Thus, all channelling constraints are AC.

To show GAC\(_{\wedge}V\) \( \not\rightarrow \) GAC\(_{\wedge}\) \( \vee \), by monotonicity, we have GAC\(_{\wedge}V\) \( \not\sim \) GAC\(_{\wedge}\). To show the reverse, consider an injection problem which is GAC\(_{\wedge}\) \( \vee \), but not GAC\(_{\wedge}\) \( \wedge \). Then there exists at least one not-equals constraint on the primal variables, say \( F_i \) and \( F_j \), that is not AC. So, \( F_i = F_j = k \) and \( i \neq j \). But, since the channelling constraints are AC, we have \( G_k = i \) and \( G_k = j \). Hence, \( i = j \). This is a contradiction as \( i \neq j \). QED.

### 6.3.3 Maintaining arc-consistency

The results with respect to arc-consistency can be lifted to algorithms that maintain (generalized) arc-consistency during search. Indeed, the gaps between the primal \textit{alldifferent} \( V \) and the channelling constraints \( c_3 \) and \( c_4 \) with the ICs or the dual not-equals constraints, as well as the primal not-equals constraints can be exponentially large. Recall that we write \( X_A \Rightarrow X_B \) iff \( X_A \rightarrow X_B \) and there is a problem on which algorithm \( X \) visits exponentially fewer branches with \( A \) than \( B \). Note that GAC\(_{\vee}\) and AC are both polynomial to enforce, so an exponential reduction in branches translates to an exponential reduction in runtime.

**Theorem 6** \textit{On an injection problem:}

\[
MGAC_{\vee} \Rightarrow MAC_{\wedge}c_3 \leftrightarrow MAC_{\wedge}c_4 \leftrightarrow MAC_{\wedge}
\]
Proof: To show $M$ takes $n$! branches, consider an injection problem with $n + 3$ variables, where $F_i = \{1, \ldots, n\}$ for all $i \leq n + 1$ and $F_{n+2} = F_{n+3} = \{n + 1, n + 2, n + 3, \ldots, m\}$, and $G_j = \{1, \ldots, n\}$ for all $j \leq m$. Then, given a lexicographical variable ordering, $M$ immediately fails, whilst $M$ takes $n$! branches.

Showing that $MAC_{\phi_{c1}} \iff MAC_{c1} \iff MAC_{\phi}$ follows immediately from the theorem $AC_{\phi_{c1}} \iff AC_{c1} \iff AC_{\phi}$. QED.

**Theorem 7** On an injection problem:

$M \Rightarrow M_{GAC_{c1}[W]} \Rightarrow MAC_{\phi_{c2},\phi_{c1}} \Rightarrow MAC_{c2} \Rightarrow MAC_{\phi}$

Proof: To show $M$ takes $n$! branches, consider an injection problem with $F_i = \{1, \ldots, n\}$ for all $i \leq n + 1$ and $F_{n+2} = F_{n+3} = \{n + 1, n + 2, n + 3, \ldots, m\}$. Then, given a lexicographical variable ordering, $M$ immediately fails, whilst $M$ takes $n$! branches.

To show $M_{GAC_{c1}[W]} \Rightarrow MAC_{\phi}$, consider an injection problem with $F_i = \{1, \ldots, n\}$ for all $i \leq n + 1$ and $F_{n+2} = F_{n+3} = \{n + 4, n + 5, \ldots, m\}$. Then, given a lexicographical variable ordering, $M$ immediately fails, whilst $M$ takes $n$! branches.

To show $MAC_{\phi_{c2},\phi_{c1}} \Rightarrow MAC_{\phi}$, consider an injection problem from $\{1, \ldots, n\}$ into $\{1, \ldots, n + 1\}$ with $F_i = \{1, \ldots, n\}$ for all $i \leq n$, $G_j = \{1, \ldots, m\}$ for all $j < n$, and $G_n = G_{n+1} = \{m\}$. Then, given a lexicographical variable ordering, $M$ immediately fails, whilst $M$ takes $n$! branches.

Showing that $MAC_{\phi_{c2},\phi_{c1}} \iff MAC_{\phi}$ follows immediately from $AC_{\phi_{c2},\phi_{c1}} \iff AC_{c2}$, and showing that $MAC_{c2} \iff MAC_{\phi}$ follows immediately from $AC_{c2} \iff AC_{\phi}$. QED.

**Theorem 8** On an injection problem:

$M \Rightarrow M_{GAC_{c1}[W]} \Rightarrow MAC_{\phi_{c3},\phi_{c1}} \Rightarrow MAC_{c3} \Rightarrow MAC_{\phi}$

Proof: To show $M$ takes $n$! branches, consider an injection problem with $F_i = \{1, \ldots, n\}$ for all $i \leq n + 1$ and $F_{n+2} = F_{n+3} = \{n + 1, n + 2, n + 3, \ldots, m\}$. Then, given a lexicographical variable ordering, $M$ immediately fails, whilst $M$ takes $n$! branches.

To show $M_{GAC_{c1}[W]} \Rightarrow MAC_{c3}$, consider an injection problem with $F_i = \{1, \ldots, n\}$ for all $i \leq n + 1$ and $F_{n+2} = F_{n+3} = \{1, \ldots, m - 2\}$. Then, given a lexicographical variable ordering, $M$ immediately fails, whilst $M$ takes $n$! branches.

Showing that $MAC_{\phi_{c3},\phi_{c1}} \iff MAC_{c3} \iff MAC_{\phi}$ follows immediately from $GAC_{\phi_{c3},\phi_{c1}} \iff GAC_{c3} \iff AC_{c3} \iff AC_{\phi}$. QED.

### 6.3.4 Forward checking

Maintaining (generalized) arc-consistency on injection problems can be expensive. We may therefore consider using a more restricted local consistency property like the one enforced by forward checking. For example, the Choco finite-domain toolkit [49] in Claire [10] uses just nFC0 [3] on the alldiffrent constraint.

For FC, we will consider neither adding the occurs constraints to the channelling constraints in $c_2$ nor adding the ICs to the channelling constraints in $c_2$ and in $c_3$. 93
We first show that the channelling constraints in $c_1$ remain as tight as the primal not-equals constraints in $\neq$ wrt FC, but not as tight as the alldifferent constraint in $\forall$. Then, we show that the channelling constraints in $c_3$ and the primal not-equals constraints in $\neq$ are not as tight as the dual not-equals and the channelling constraints in $c_2$ wrt FC, which are not as tight as the alldifferent constraint in $\forall$. Finally, we show that the channelling constraints in $c_3$ are as tight as the primal not-equals constraints in $\neq$, but not as tight as the alldifferent constraint in $\forall$.

**Theorem 9** On an injection problem:

$$nFC_2 \rightarrow FC_{\neq c_1} \iff FC_{c_1} \iff FC_{\neq} \rightarrow nFC_0$$

Proof: [37] proves that $FC_{\neq}$ implies $nFC_0$, and that $nFC_1$ implies $FC_{\neq}$. [81] extends the results on permutation problems and shows that $FC_{\neq} \rightarrow nFC_0$ and $nFC_1 \rightarrow FC_{\neq}$. [3] proves $nFC_2$ implies $nFC_1$ and [81] extends the result to show that $nFC_2 \rightarrow nFC_1$ for permutation problems. The results in [81] also hold for injection problems because we use the same alldifferent constraint.

To show $FC_{c_1} \iff FC_{\neq}$, consider assigning the value $j$ to the primal variable $F_i$. $FC_{\neq}$ removes $j$ from the domain of all other primal variables except $F_i$. $FC_{c_1}$ instantiates the dual variable $G_j$ with the value $i$, and then removes $j$ from the domain of all other primal variables, except $F_i$. Hence $FC_{\neq}$ prunes all the values that $FC_{c_1}$ prunes. Furthermore, when assigning $j$ to $F_i$ leads to a domain wipeout of another primal variable, say $F_k$, because of the binary not-equals constraint, the same domain wipeout would happen to $G_j$ because $G_j$ would have no value $i$ supported due to the channelling constraint between $F_k$ and $G_j$. Hence, $FC_{c_1}$ has a domain wipeout whenever $FC_{\neq}$ does.

To show $nFC_2 \rightarrow FC_{\neq c_1}$, consider instantiating the primal variable $F_i$ with the value $j$. $FC_{\neq c_1}$ removes $j$ from the domain of all primal variables except $F_i$ and instantiates $G_j$ with the value $i$. $nFC_2$ also removes $j$ from the domain of all primal variables except $F_i$. The instantiation of $G_j$ with the value $i$ does not lead to any further pruning. Thus, $nFC_2$ has a domain wipeout whenever $FC_{\neq c_1}$ does. To show strictness, consider a 4 variable injection problem with $F_1 = F_2 = F_3 = F_4 = \{1, 2, 3\}$ and $G_1 = G_2 = G_3 = G_4 = G_5 = \{1, 2, 3, 4\}$. Irrespective of the variable and value ordering, $FC_{\neq c_1}$ takes 6 branches to show the problem is unsatisfiable and $nFC_2$ by comparison takes 3 branches. QED.

**Theorem 10** On an injection problem:

$$nFC_2 \rightarrow FC_{\neq c_2 \neq} \iff FC_{c_2 \neq} \rightarrow FC_{c_2} \iff FC_{\neq}$$

Proof: To show $nFC_2 \rightarrow FC_{\neq c_2 \neq}$, consider assigning the value $j$ to the primal variable $F_i$. $FC_{\neq c_2 \neq}$ removes $j$ from the domains of all primal variables except $F_i$, removes $i$ from the domains of all dual variables except $G_j$, and instantiates $G_j$ with the value $i$. $nFC_2$ also removes $j$ from the domains of all primal variables except $F_i$. The only possible difference is if one of the other dual variables, say $G_i$, has a domain wipeout. But this cannot be possible because at most $n$ values (the number of primal variables)
Irrespective of the variable and value ordering, FC\(_c\) has a domain wipeout whenever FC\(_{\bar{c} \neq \bar{d}}\) does. To show strictness, consider a 7 variable injection problem with \(F_1 = F_2 = F_3 = F_4 = \{1, 2, 3\}\) and \(F_5 = F_6 = F_7 = \{4, 5, 6, 7, 8\}\). Irrespective of the variable and value ordering, FC\(_{\bar{c} \neq \bar{d}}\) takes at least 7 branches to show the problem is unsatisfiable while nFC2\(_G\) takes no more than 5 branches.

To show FC\(_{\overline{c} \neq \bar{d}}\) ⊃ FC\(_{\bar{c}}\), by monotonicity, we have FC\(_{\bar{c} \neq \bar{d}}\) implies FC\(_{\bar{c}}\). To show strictness, consider a 3 variable injection problem in which \(F_1 = F_2 = F_3 = \{1, 2\}\), \(G_1 = G_2 = \{1, 2, 3, 4\}\), and \(G_3 = G_4 = \{4\}\). FC\(_{\bar{c}}\) takes 4 branches to show that the problem is unsatisfiable while FC\(_{\bar{c} \neq \bar{d}}\) only takes 2 branches.

To show FC\(_{\bar{c}}\) ⊃ FC\(_{\bar{d}}\), consider assigning the value \(j\) to the primal variable \(F_i\). FC\(_{\bar{d}}\) removes \(j\) from the domain of all other primal variables except \(F_i\). FC\(_{\bar{c}}\) instantiates the dual variable \(G_j\) with the value \(i\), removes \(j\) from the domain of all other primal variables except \(F_i\), and removes \(i\) from the domains of all dual variables except \(G_j\). The only possible difference is if a dual variable, say \(G_i\) has a domain wipeout. But this cannot be possible because at most \(n\) values (the number of primal variables) can be pruned from the domain of \(G_i\), which originally is of size \(m\). Hence FC\(_{\bar{d}}\) prunes all the values that FC\(_{\bar{c}}\) prunes. Furthermore, if assigning \(j\) to \(F_i\) leads to a domain wipeout of another primal variable, say \(F_k\), because of the binary not-equals constraint \((F_i \neq F_k)\), then because of the channelling constraint between \(F_k\) and \(G_j\), the variable \(G_j\) must have value \(k\) in its domain, which leads to a domain wipeout (because we have a domain wipeout of \(F_k\)). Hence, FC\(_{\bar{c}}\) has a domain wipeout whenever FC\(_{\bar{d}}\) does. QED.

**Theorem 11** On an injection problem:

\[
n\text{FC}_2 \Rightarrow \text{FC}_{\bar{c}} \Rightarrow \text{FC}_{\bar{d}}
\]

**Proof:** To show nFC2\(_G\) ⊃ FC\(_{\bar{c}}\), consider assigning the value \(j\) to the primal variable \(F_i\). FC\(_{\bar{c}}\) removes \(j\) from the domains of all primal variables except \(F_i\), removes \(i\) from the domains of all dual variables except \(G_j\), and instantiates \(G_j\) with the value \(i\). nFC2\(_G\) also removes \(j\) from the domains of all primal variables except \(F_i\). The only possible difference is if one of the other dual variables, say \(G_i\) has a domain wipeout. But this cannot be possible because at most \(n\) values (the number of primal variables) can be pruned from the domain of \(G_i\), which originally has domain of size \(n + 1\). Hence nFC2\(_G\) has a domain wipeout whenever FC\(_{\bar{c}}\) does. To show strictness, consider a 7 variable injection problem with \(F_1 = F_2 = F_3 = F_4 = \{1, 2, 3\}\) and \(F_5 = F_6 = F_7 = \{4, 5, 6, 7, 8\}\). Irrespective of the variable and value ordering, FC\(_{\bar{c}}\) takes at least 7 branches to show the problem is unsatisfiable while nFC2\(_G\) takes no more than 5 branches.

To show FC\(_{\bar{c}}\) ⊃ FC\(_{\bar{d}}\), consider assigning the value \(j\) to the primal variable \(F_i\). FC\(_{\bar{d}}\) removes \(j\) from the domain of all other primal variables except \(F_i\). FC\(_{\bar{c}}\) instantiates the dual variable \(G_j\) with the value \(i\), removes \(j\) from the domain of all other primal variables except \(F_i\), and removes \(i\) from the domains of all dual variables except \(G_j\). The only possible difference is if a dual variable, say \(G_i\) has a domain wipeout. But this is impossible because at most \(n\) values (the number of primal variables) can be pruned from the domain of \(G_i\), which originally is of size \(n + 1\). Hence FC\(_{\bar{d}}\) prunes all the values that FC\(_{\bar{c}}\) prunes. Furthermore, if assigning \(j\) to \(F_i\) leads to a domain wipeout of another primal variable, say \(F_k\), because of the binary not-equals constraint \((F_i \neq F_k)\), then because of the channelling constraint between \(F_k\) and \(G_j\), we will have a domain wipeout of the variable \(G_j\). Hence, FC\(_{\bar{c}}\) has a domain wipeout whenever FC\(_{\bar{d}}\) does. QED.
6.3.5 Bounds consistency

Another common method to reduce costs is to enforce just bounds consistency. Bounds consistency is defined for ordered domains and a domain will be denoted by an interval [min, max] where min is the minimum value in the domain and max is the maximum value in the domain. With bounds consistency on injection problems, we obtain the same ordering of the models as with arc-consistency.

Theorem 12 On an injection problem:

\[ BC_y \Rightarrow BC_{\neq c_1} \iff BC_{c_1} \iff BC_{\neq} \uparrow AC_{\neq} \]

Proof: To show BC_{\neq c_1} \iff BC_{\neq}, consider an injection problem which is BC_{\neq}, but one of the primal not-equals constraints is not BC. Then, it would involve two variables, F_i and F_j, both with identical interval domains, [k, k]. Enforcing BC on the channelling constraint between F_i and G_k would reduce G_k to the domain \([i, i]\). Enforcing BC on the channelling constraint between F_j and G_k would then cause a domain wipeout. But this contradicts the channelling constraints being BC. Hence all the primal not-equals constraints must be BC. Consider now an injection problem which is BC_{\neq} but one of the channelling constraints, say F_i = k \rightarrow G_k = i, is not BC. That is, F_i has interval domain [k, k] and G_k has i removed from its domain. For G_k to have i not supported, another primal variable, say F_j, should have interval domain [k, k]. Thus F_i and F_j have the same interval domain [k, k], which is a contradiction as the not-equals constraints are BC. Hence all channelling constraints are BC.

To show BC_y \rightarrow BC_{\neq c_1}, consider an injection problem which is BC_y. The not-equals constraints are trivially BC. Now since BC_{c_1} \rightarrow BC_{\neq}, the channelling constraints are BC. Thus the problem is BC_{\neq c_1}. To show strictness, consider a 3 variable injection problem with F_1 = F_2 = F_3 = [1, 2] and G_1 = G_2 = G_3 = G_4 = [1, 4]. This is BC_{\neq c_1} but not BC_y. QED.

Theorem 13 On an injection problem:

\[ BC_y \rightarrow BC_{\neq c_2|W} \iff BC_{c_2|W} \rightarrow BC_{\neq c_2|d} \iff BC_{c_2|d} \rightarrow BC_{\neq} \uparrow AC_{\neq} \]

Proof: To show BC_y \rightarrow BC_{c_2|W}, consider an injection problem which is BC_y. Now assume that the ICs are not BC. Then, there must exist, at least, m-n+1 dual variables with domains not in D. Because the channelling constraints are BC, m-n+1 primal variables should have n values pruned from their domains. Thus, m-n+1 primal variables have the same domain size m-n. But, since the all different constraint is BC, this is a contradiction. So, the ICs also are BC. To show strictness, consider an injection problem with F_1 = F_2 = F_3 = [1, 2], F_4 = [1, 6], G_1 = G_2 = [1, 6], and G_3 = G_4 = G_5 = G_6 = [4, 6]. This is BC_{c_2|W} but not BC_y.

To show BC_{c_2|W} \rightarrow BC_{\neq c_2|d}, consider an injection problem which is BC_{c_2|W}. Suppose the not-equals constraint between G_i and G_j, was not BC. Then, in the first case, G_i = G_j = [k, k] and k < n+1, which is impossible because the channelling constraints F_1 = i \leftrightarrow G_i = k and F_1 = j \leftrightarrow G_j = k are BC. In the second case, k would be greater
than $n$, which is impossible by construction of the primal and dual model. To show strictness, consider an injection problem with $F_1 = F_2 = F_3 = [1,2]$, $G_1 = G_2 = [1,5]$, and $G_3 = G_4 = G_5 = [4,5]$. This is $BC_{c_3 \neq d} w$, but not $BC_{c_2 \neq d} w$.

To show $BC_{c_2 \neq d} \rightarrow BC_{c_2}$, by monotonicity, we have $BC_{c_2 \neq d} \Rightarrow BC_{c_2}$. To show strictness, consider an injection problem with $F_1 = F_2 = F_3 = [1,2]$, $G_1 = G_2 = [1,4]$, and $G_3 = G_4 = [4,4]$. This is $BC_{c_2} w$, but not $BC_{c_2} w$.

To show $BC_{c_2} \leftrightarrow BC_{\phi}$, consider an injection problem which is $BC_{c_2}$ but one of the primal not-equals constraints is not BC. Then, it would involve two variables, $F_i$ and $F_j$ both with identical interval domains, $[k, k]$. Enforcing BC on the channelling constraint between $F_i$ and $G_k$ would reduce $G_k$ to the domain $[i, i]$. Enforcing BC on the channelling constraint between $F_j$ and $G_k$ would then cause a domain wipeout. But this contradicts the channelling constraints being BC. Hence all the primal not-equals constraints are BC.

Consider now an injection problem which is $BC_{\phi}$ but one of the channelling constraints is not BC. Then, it would involve two variables, $G_i$ and $G_j$, and $F_k$ has $i$ removed from its domain. For $G_k$ to have $i$ not supported, another primal variable, say $F_j$, should have interval domain $[k, k]$. Thus $F_i$ and $F_j$ has the same interval domain $[k, k]$, which is a contradiction as the not-equals constraints are BC. Hence all channelling constraints are BC. QED.

**Theorem 14** On an injection problem:

$$BC_{\phi} \rightarrow BC_{/ c_3 \neq 0} \iff BC_{c_3 | w} \rightarrow BC_{\neq c_0} \iff BC_{c_0} \leftrightarrow BC_{c_2} \leftrightarrow BC_{\phi}$$

**Proof:** To show $BC_{\phi} \rightarrow BC_{c_3 | w}$, consider an injection problem which is $BC_{\phi}$. Now assume that the ICs are not BC. Then, there must exist, at least, $m - n + 1$ dual variables with domain $[n + 1, n + 1]$. Because the channelling constraints are BC, $m - n + 1$ primal variables should have $n$ values pruned from their domains. Thus, $m - n + 1$ primal variables have the same domain size $m - n$. But, since the all-different constraint is BC, this is a contradiction. So, the ICs also are BC. To show strictness, consider an injection problem with $F_1 = F_2 = F_3 = [1,2]$, $F_4 = [1,5]$, $G_1 = G_2 = [1,5]$, and $G_3 = G_4 = G_5 = [4,5]$. This is $BC_{c_3 | w}$ but not $BC_{\phi}$.

To show $BC_{\neq c_0} \leftrightarrow BC_{c_3}$, by monotonicity, we have $BC_{\neq c_0} \Rightarrow BC_{c_3}$. To show the reverse, suppose that all the channelling constraints are BC. Consider an occurs constraint that is not BC. Then there exist (at least) two dual variables, say $G_i$ and $G_j$ and $i \neq j$, that have the same domain $[k, k]$ and $k \leq n$. But since the channelling constraints are BC, we have $F_i = [i, i] = [j, j]$. Hence, $i = j$. This is a contradiction. So, all occurs constraints are BC.

To show $BC_{c_3 | w} \rightarrow BC_{c_3}$, by monotonicity, we have $BC_{c_3 | w} \Rightarrow BC_{c_3}$. To show strictness, consider an injection problem with $F_1 = F_2 = F_3 = [1,2]$, $G_1 = G_2 = [1,4]$, and $G_3 = G_4 = G_5 = [4,5]$. This is $BC_{c_3}$ but not $BC_{c_3 | w}$.

To show $BC_{c_3} \leftrightarrow BC_{\phi}$, consider an injection problem which is $BC_{c_3}$ but one of the primal not-equals constraints is not BC. Then, it would involve two variables, $F_i$ and $F_j$, both with identical interval domains, $[k, k]$. Enforcing BC on the channelling constraint between $F_i$ and $G_k$ would reduce $G_k$ to the domain $[i, i]$. Enforcing BC on the channelling constraint between $F_j$ and $G_k$ would then cause a domain wipeout. But this contradicts the channelling constraints being BC. Hence all the primal not-equals constraints are BC. Consider now an injection problem which is $BC_{\phi}$ but one of the channelling constraints,
say $F_i = k \leftrightarrow G_k = i$, is not BC. That is, $F_i$ has interval domain $[k, k]$ and $G_k$ has $i$ removed from its domain. For $G_k$ to have $i$ not supported, another primal variable, say $F_j$, should have interval domain $[k, k]$. Thus $F_i$ and $F_j$ has the same interval domain $[k, k]$, which is a contradiction as the not-equals constraints are BC. Hence all channelling constraints are BC. QED.

6.4 Asymptotic comparison

The previous results compare the different models with respect to the amount of pruning achieved. With respect to arc-consistency, we have shown that the occurs constraints are redundant. Therefore, their addition will only increase the run-time without achieving any more pruning. The models incorporating the occurs constraints can be safely discarded. However, we can add details to these results by comparing the asymptotic behavior of the more interesting ones, namely the relative costs of achieving GAC, AC$_{c_1}$, GAC$_{c_2|W|}$, AC$_{c_2\neq d}$, AC$_{c_2}$, AC$_{c_3|W|}$, AC$_{c_3}$, and AC$_{\neq}$:

- Régis’s algorithm [64] achieves GAC$_{\neq}$ in $O(n^2m^2)$, where $n$ is the number of variables and $m$ is their domain size.
- AC on binary constraints can be achieved in $O(ed^2)$, where $e$ is the number of constraints and $d$ is their domain size. As there are $O(nm)$ channelling constraints, AC$_{c_1}$, AC$_{c_2}$, and AC$_{c_3}$ naively takes $O(nm^3)$ time. However, by taking advantage of the functional nature of channelling constraints, we can reduce this to $O(nm^2)$ for $c_2$ and $c_3$ and $O(nm)$ for $c_1$ using the AC-5 algorithm of [77]. Similarly, there are $O(m)$ channelling constraints between the set variable and the dual variables, so naively arc consistency on these constraints is achieved in $O(m^3)$ but can be achieved in $O(m^2)$ by taking advantage of the semantics of the constraints.
- The ICs can be treated as a special case of the global cardinality constraint [62] with time complexity $O(c^2d)$, where $e$ is the number of variables and $d$ is their domain size. Since the domains are Booleans, we achieve GAC$_{|W|}$ in $O(m^2)$.
- AC$_{\neq}$ also naively takes $O(n^4)$ time for the primal variables (respectively, $O(m^4)$ time for the dual variables) as there are $O(n^2)$ binary not-equals constraints (respectively $O(m^2)$ binary not-equals constraints on the dual variables). However, we can take advantage of the special nature of the binary not-equals constraint to reduce this to $O(n^2)$ (respectively $O(m^2)$ for the dual variables) with a careful implementation as each not-equals constraint needs to be made AC just once.

We proved in Theorem 3 that GAC$_{\neq} \rightarrow$ AC$_{c_1} \leftrightarrow$ AC$_{\neq}$ and their costs are $O(n^3m^2)$, $O(nm)$, and $O(n^2m^2)$ respectively. Asymptotic analysis shows that enforcing AC$_{c_1}$ has asymptotically slightly more cost than enforcing AC$_{\neq}$. However, having the dual variables could be advantageous in conjunction with variable and value ordering heuristics.

We also proved in Theorem 4 that GAC$_{\neq} \rightarrow$ GAC$_{c_2|W|} \rightarrow$ AC$_{c_2\neq d} \rightarrow$ AC$_{c_2} \leftrightarrow$ AC$_{\neq}$ and their costs are $O(n^3m^2)$, $O(nm^2)$, $O(n \times m^2)$, $O(nm^3)$, and $O(n^2)$ respectively. Asymptotic analysis shows that the channelling constraints are more costly than the not-equals constraints and bring no more pruning. When we add not-equals constraints on the dual variables, the overall asymptotic cost is still the same as the channelling constraints alone, but we achieve more pruning. By adding the ICs, the asymptotic cost is still the same.
as with the channelling constraints, but we achieve even more pruning when compared to adding not-equals constraints on the dual variables.

We showed in Theorem 5 that $\text{GAC}_\forall \rightarrow \text{GAC}_{c_3}|_W \rightarrow \text{AC}_{c_3} \leftrightarrow \text{AC}_\forall$ and their costs are $O(n^2m^2)$, $O(nm^2)$, $O(nm^2)$, and $O(n^2)$ respectively. Again, asymptotic analysis shows that channelling constraints are more costly than the not-equals constraints and bring more pruning, with an extra cost, only when the cardinality constraint is added. Maintaining generalized arc-consistency on the $\text{alldifferent}$ constraint is the most costly.

From the theoretical and asymptotic analysis, we conclude that, for an injection problem:

- The channelling constraints of model $c_1$ achieve the same amount of pruning as the binary not-equals constraints on model $\neq$, but less pruning than the $\text{alldifferent}$ constraint of the model $\forall$. Their respective costs reflect that achieving GAC on the $\text{alldifferent}$ constraint is more costly than achieving AC on the channelling constraints, which is more costly than achieving AC on the binary not-equals constraints. However, we might still want to keep model $c_1$ because it might be useful for developing cheap value ordering heuristics.

- The channelling constraints of model $c_2$ when used in conjunction with the dual not-equals constraints of model $\neq$ or the ICs, achieve more pruning than the binary not-equals constraints on the primal variables of model $\neq$, but less pruning than an $\text{alldifferent}$ constraint of the $\forall$ model. Their respective costs also reflect that achieving GAC on the $\text{alldifferent}$ constraint is more costly than achieving GAC on the channelling and ICs (or dual not-equals constraints), which is more costly than achieving AC on the binary not-equals constraints. We can safely discard the model $c_2$ and $c_2 \neq$ because their constraints achieve less pruning than the constraints of model $c_2|_W$ at the same cost.

- The channelling constraints of model $c_3$ when used in conjunction with the ICs, achieve more pruning than the binary not-equals constraints on the primal variables of model $\neq$, but less pruning than an $\text{alldifferent}$ constraint of the $\forall$ model. Their respective costs also reflect that achieving GAC on the $\text{alldifferent}$ constraint is more costly than achieving GAC on the channelling and ICs, which is more costly than achieving AC on the binary not-equals constraints. We can safely discard the model $c_3$ because its constraints achieve less pruning than the constraints of model $c_3|_W$ at the same cost.

### 6.5 Experimental comparison

After the theoretical and asymptotic analysis ruled out some models such as the one that uses the $\text{occur}$ constraint. We now ran some empirical study on three injection problems to measure the trade-off between the amount of pruning to be achieved and the amount of time to achieve that. We ran experiments using the SICSTUS PROLOG [9] finite domain constraint library and opl[75]. We consider the models that survived the theoretical and

---

2Developing a variable ordering heuristic on the dual variables corresponds to a value ordering heuristic on the primal variables. Since, in the dual model the values are explicitly represented as variables, it may be cheaper to develop a value ordering heuristic on the primal variables by developing a variable heuristic on the dual variables. The channelling constraints will trigger the changes back and forth between the primal and dual variables.
Modified $n$-Queens problem

In the modified $n$-Queens problem, we assume we have more columns than rows. We seek an injection from the set of queens $Q = \{1, \ldots, n\}$ into the set of columns $C = \{1, \ldots, m\}$, where $m > n$. The constraints are that no queen should attack any other queen. We ran three sets of experiments using opl and measure the number of branches visited to find one solution.

The results of the first set, when $m = n + 1$, are shown in Figure 6.11. Note that when $m = n + 1$, the models $c_2$ and $c_3$ are the same. We notice that MGAC$_{c_2\mid W\mid}$ visits more branches than MGAC$_v$, but less branches than MAC$_\neq$. MGAC$_v$ is the fastest solution method, but the channelling constraints are naively handled by opl. Efficiently implementing the channelling constraints goes beyond the scope of this thesis. So, we conjecture that if we carefully implement the channelling constraints, then MGAC$_{c_2\mid W\mid}$ will be very competitive with MGAC$_v$. The results of the second set, when $m = n + 2$, are shown in Figure 6.12. We notice that MGAC$_{c_2\mid W\mid}$ and MGAC$_{c_3\mid W\mid}$ visit the same number of branches, but visit less branches than MGAC$_v$ while MAC$_\neq$ records the largest number of visited branches. We again conjecture that if we carefully implement the channelling constraints, then MGAC$_{c_2\mid W\mid}$ and MGAC$_{c_3\mid W\mid}$ will be very competitive with MGAC$_v$. The results of the last set, when $m = n + 3$, are shown in Figure 6.13. The results are similar to the results of the second set.

<table>
<thead>
<tr>
<th>$n$</th>
<th>MGAC$_v$</th>
<th>MGAC$_{c_2\mid W\mid}$</th>
<th>MGAC$_{c_3\mid W\mid}$</th>
<th>MAC$_\neq$</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>9942</td>
<td>12698</td>
<td>26154</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>6244</td>
<td>8186</td>
<td>16651</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>143644</td>
<td>186009</td>
<td>336104</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>21140</td>
<td>29611</td>
<td>70866</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>412195</td>
<td>516356</td>
<td>972893</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.11: Branches to find one solution to the modified $n$-Queens problem where $m = n + 1$

<table>
<thead>
<tr>
<th>$n$</th>
<th>MGAC$_v$</th>
<th>MGAC$_{c_2\mid W\mid}$</th>
<th>MGAC$_{c_3\mid W\mid}$</th>
<th>MAC$_\neq$</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>7471</td>
<td>10145</td>
<td>10145</td>
<td>16128</td>
</tr>
<tr>
<td>26</td>
<td>4471</td>
<td>7449</td>
<td>7449</td>
<td>14267</td>
</tr>
<tr>
<td>27</td>
<td>27696</td>
<td>40221</td>
<td>40221</td>
<td>70924</td>
</tr>
<tr>
<td>28</td>
<td>56927</td>
<td>77662</td>
<td>77662</td>
<td>131348</td>
</tr>
<tr>
<td>29</td>
<td>171905</td>
<td>220730</td>
<td>220730</td>
<td>415438</td>
</tr>
</tbody>
</table>

Figure 6.12: Branches to find one solution to the modified $n$-Queens problem where $m = n + 2$

asymptotic analysis, namely, the model $\neq$, the model $c_2\mid W\mid$, the model $c_3\mid W\mid$, and the model $\forall$. For each injection problem, the additional constraints of the problem are stated in the same way in all models.
<table>
<thead>
<tr>
<th>$n$</th>
<th>$\text{MGAC}_v$</th>
<th>$\text{MGAC}_{c\omega}$</th>
<th>$\text{MAC}_{c\omega}$</th>
<th>$\text{MAC}_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>1440</td>
<td>2157</td>
<td>2157</td>
<td>4251</td>
</tr>
<tr>
<td>26</td>
<td>18644</td>
<td>29001</td>
<td>29001</td>
<td>46652</td>
</tr>
<tr>
<td>27</td>
<td>2341</td>
<td>3448</td>
<td>3448</td>
<td>8418</td>
</tr>
<tr>
<td>28</td>
<td>7394</td>
<td>9619</td>
<td>9619</td>
<td>26205</td>
</tr>
<tr>
<td>29</td>
<td>11063</td>
<td>17205</td>
<td>17205</td>
<td>40685</td>
</tr>
</tbody>
</table>

Figure 6.13: Branches to find one solution to the modified $n$-Queens problem where $m = n + 3$

<table>
<thead>
<tr>
<th>Period</th>
<th>Week1</th>
<th>Week2</th>
<th>Week3</th>
<th>Week4</th>
<th>Week5</th>
<th>Week6</th>
<th>Week7</th>
<th>Week8</th>
<th>Week9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Period1</td>
<td>(1.5)</td>
<td>(1.6)</td>
<td>(2.6)</td>
<td>(2.7)</td>
<td>(3.7)</td>
<td>(3.8)</td>
<td>(4.8)</td>
<td>(4.9)</td>
<td>(5.9)</td>
</tr>
<tr>
<td>Period2</td>
<td>(2.3)</td>
<td>(2.9)</td>
<td>(1.4)</td>
<td>(1.8)</td>
<td>(4.6)</td>
<td>(5.7)</td>
<td>(3.9)</td>
<td>(5.8)</td>
<td>(6.7)</td>
</tr>
<tr>
<td>Period3</td>
<td>(4.7)</td>
<td>(7.8)</td>
<td>(3.5)</td>
<td>(6.9)</td>
<td>(8.9)</td>
<td>(2.4)</td>
<td>(5.6)</td>
<td>(1.2)</td>
<td>(1.3)</td>
</tr>
<tr>
<td>Period4</td>
<td>(6.8)</td>
<td>(3.4)</td>
<td>(7.9)</td>
<td>(4.5)</td>
<td>(2.5)</td>
<td>(1.9)</td>
<td>(1.7)</td>
<td>(3.6)</td>
<td>(2.8)</td>
</tr>
</tbody>
</table>

Figure 6.14: Solution to the Sport Scheduling problem for 9 teams.

### Sport scheduling

The Sport Scheduling problem [79] consists of scheduling games between $n$ teams over $n - 1$ weeks when $n$ is even ($n$ weeks when $n$ is odd). Each week is divided into $n/2$ periods when $n$ is even ($(n - 1)/2$ periods when $n$ is odd). Each game is composed of two slots, “home” and “away”, where one team plays “home” and the other team plays “away”. The objective is to schedule a game for each period of every week such that:

- Every team plays against every other team.
- A team plays exactly once a week when we have an even number of weeks, and at most once a week when we have an odd number of teams.
- A team plays at most twice in the same period over the course of the season.

A solution to the problem when $n$ is 9 is shown in Figure 6.14.

When $n$ is odd, the Sport Scheduling problem can be modelled as follows. The set of teams can be represented by $T = \{1, \ldots, n\}$, the set of weeks can be represented by $W = \{1, \ldots, n\}$, the set of periods by $P = \{1, \ldots, (n - 1)/2\}$, and the set of slots representing “home” and “away” can be represented by the set $S = \{1, 2\}$, where 1 means “home” and 2 means “away”. Finding a schedule amounts to finding, for each week, an injection from $PS$ into $T$ where $PS$ is $P \times S$ and such that all the other constraints of the problem are satisfied.

We implemented some models of the tournament scheduling problem, in opl. We first present the number of visited branches achieved by different models of the injection in Figure 6.15. Since $|PS| = |T| - 1$, the models $c_{v}\omega$ and $c_{v}|W|$ are the same. We notice that $\text{MGAC}_{c\omega}$ visits less branches than $\text{MAC}_{c\omega}$, but more than $\text{MGAC}_v$. We conjecture that, if carefully implemented, $\text{MGAC}_{c\omega}$ is very competitive with $\text{MGAC}_v$.

When $n$ is even, the problem can be viewed as a bijection from $PS$ into $T$. We ran some experiments using the opl model given in [79] to compare with the injective model $\forall$. $\text{MGAC}_v$ visits the least number of branches in the Figure 6.15. We perform this experiment because when $n$ is odd, we could add a dummy team and transform
Figure 6.15: Branches to compute one solution to the Sport Scheduling with odd number of teams

<table>
<thead>
<tr>
<th>n</th>
<th>MGAC__v</th>
<th>MGAC___W</th>
<th>MAC__3</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>171</td>
<td>212</td>
<td>216</td>
</tr>
<tr>
<td>11</td>
<td>10211</td>
<td>12673</td>
<td>15316</td>
</tr>
<tr>
<td>13</td>
<td>60359</td>
<td>73182</td>
<td>91316</td>
</tr>
</tbody>
</table>

Figure 6.16: Branches to compute one solution to the Sport Scheduling problem using a bijective model and an injective model

<table>
<thead>
<tr>
<th>n</th>
<th>injection model</th>
<th>bijection model for n + 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>517</td>
</tr>
<tr>
<td>9</td>
<td>171</td>
<td>&gt; 272141</td>
</tr>
<tr>
<td>11</td>
<td>10211</td>
<td>&gt; 343960</td>
</tr>
</tbody>
</table>

the problem into a bijection. We use the bijection model in [79] where the bijectiveness constraint is enforced using a global \textit{alldifferent} constraint. Walsh has shown that on bijections, the model using \textit{alldifferent} visits the least number of branches [81]. We use the same labelling strategy (lexicographical ordering for the variables and numeric ordering for the values) for both the injection model \forall and the bijection model in [79]. The results are shown in Figure 6.16. We notice from Figure 6.16 that the injective model outperformed the bijective one. We notice that for the sport scheduling problem researchers usually model the problem assuming \( n \) is even and then add an extra team when \( n \) is odd. However, we see that the injective models is also competitive and hence, suggest that an injection model may be used to model the problem for an odd number of teams rather than adding a dummy team and using a bijection model.

**Graceful graphs**

Graceful graphs have many applications in radio astronomy and cryptography [60]. Given a graph \( G = (V, E) \) with \( n = |V| \) vertices and \( m = |E| \) edges, the graph is said to be graceful if \( n < m \) and there are unique vertex labels \( f : V \rightarrow \{0, \ldots, m\} \) (i.e., \( f \) is injective) and unique edge labels \( g : E \rightarrow \{1, \ldots, m\} \) (i.e., \( g \) is bijective) such that \( g(i,j) = \left| f(i) - f(j) \right| \) for each edge \( e \in E \) between vertex \( i \) and \( j \). It is very difficult to show that complete graphs are not graceful as the number of vertices increases. In our experiments, we tested the models \( \forall, c_2 \neq d, c_2|W|, \) and \( \forall \) on complete graphs (which are proven to be not graceful), and the results are shown in Figure 6.17, as well as for 3 random instances that are graceful, using the SICSTUS PROLOG finite domain constraint solver, and the results are shown in Figure 6.18. On all instances, MAC\_\_v records the same number of backtracks as all the other models, including MGAC\_\_v, which results in MAC\_\_v being the quickest solution method. We conjecture that this is due to the graceful graph problem being very loose.
6.6 Summary of the results

We have proposed different models of injection problems and used the constraint tightness parameterized by the level of local consistency being enforced, as proposed by Walsh [81], to compare these models.

We proved that, with respect to arc-consistency, forward checking, and bounds consistency, a single primal all-different constraint is tighter than channelling constraints together with the implied or the dual not-equals constraints, but that the channelling constraints alone are as tight as the primal not-equals constraints. Both these gaps can lead to an exponential reduction in search cost when MAC or MGAC are used. The theoretical results showed that the occurs constraints are redundant, so we can safely discard them.

The asymptotic analysis added details to the theoretical results. We conclude that it is safe to discard the model $c_2$ and $c_2 \neq d$ because they achieve less pruning than model $c_2[W]$ at the same cost. Similarly, we can discard the model $c_3$ because it achieves less pruning than model $c_3[W]$ at the same cost. However, we keep model $c_1$ even though it achieves the same amount of pruning as $\neq$ at a higher cost because it might allow the development of cheap value ordering heuristics.

Experimental results on the modified $n$-Queens problem and the Sport Scheduling problem confirmed that MGAC on channelling and implied constraints outperformed MAC on primal not-equals constraints, and could be competitive with maintaining GAC on a primal all-different constraint. However, on less constrained problems such as the Graceful Graph problem and when the size of the target $m$ is considerably larger than the size $n$ of the source set of the injection as shown in Figure 6.17 and Figure 6.18, maintaining AC on the primal not-equals constraints achieves comparable results with maintaining GAC on the primal all-different constraint. Hence it might not be beneficial to have additional constraint propagation for such cases.

Walsh theoretically compared three models of permutation problems. A primal model using a global all-different constraint, a primal model using binary not-equal constraints, and a minimal integrated model that uses channelling constraints. He showed that, with respect to arc-consistency, forward checking, and bounds consistency, a single primal all-different constraint is tighter than channelling constraints, but that channelling con-
straints are tighter than primal not-equals constraints. However, our results show that, for injections, the channelling constraints are tighter than the primal not-equals constraints only when either the dual not-equals constraints or the ICs are added, but still not as tight as the primal all-different constraint.

6.7 Summary

In this chapter, we exposed some of the elements that make the task of model selection difficult. Then, we proposed a theoretical analysis of different models of injection problems. We supported the theoretical results with an asymptotic analysis and an empirical study on three different problems.

The theoretical approach, which is based on the measure proposed by Walsh [81], is not very accurate and ignores the contribution of other constraints in the problem. However, the presented results provide a significant step towards solving the question of automatic selection of models. For instance, the theoretical approach in some cases rules out models that are a-priori known to be worse than others. The injective model that incorporates the occurs constraint is such a case. The asymptotic analysis rules out models that achieve the same amount of pruning as other models but at a higher cost. The injective model \( c_2 \neq d \) is such a case. As more work is done in this direction, the pieces of the puzzle will start becoming more apparent.

The contributions of this chapter can be summarized as follows:

- an analysis of the task of model selection;
- a theoretical comparison of different models of injection problems supported by an asymptotic analysis and an empirical study;

In the following chapter, we will propose a practical modelling tool that would incorporate the findings established in Chapter 4, Chapter 5, and this chapter.
Chapter 7

A Practical Modelling Tool

In this chapter, we propose a practical modelling tool that is based on the contributions presented throughout the thesis. The modelling tool, called Fiona, helps users to explore alternate Ł models of function problems expressed at a suitable level of abstraction (the $\mathcal{F}$ level). The generated Ł models are independent of any target CP language, but by fixing the target CP language, the Ł models, supported in that language, can be automatically generated as the problem now is just a syntactic one. This is due to the fact that in an Ł model we have made all the necessary modelling decisions. An Ł model describes the variables and their domains, as well as the constraints and the cost function that need to be stated on these variables. However, it is the user’s responsibility to actually make a decision on which Ł model to use for her function problem.

The rest of the chapter is organized as follows. In Section 7.1, the grammar of the input language to the tool is presented, as well as the requirements that should be met by the tool. Then, the architecture of the tool is presented in Section 7.2, and the different components are related to different parts of the thesis. We also assess the scope of the $\mathcal{F}$ language and the quality of the generated models in Section 7.3 and provide some extensions to the tool in Section 7.4. Finally, we summarize in Section 7.5.

7.1 Design decisions

We present the syntax of the input language, $\mathcal{F}$, and argue why to make this a specification language and not a programming language.

7.1.1 Syntax

A function model consists of a declaration part, followed by an optional optimization part, and a constraint part, as described next.

In the declaration part, typed inputs and decision variables are declared. The primitive types are the integers (int) and integer sets (set(int)). The type constructors are the cross-product constructor (written $\times$ and used in an infix way) for sets of tuples, binary constructors (written $\rightarrow$, $\leftrightarrow$ and used in an infix way) for functions between sets, a unary constructor (perm) for permutations of sets, a binary constructor (seq) for sequences of fixed length over sets, and a binary constructor (bseq) for sequences of bounded length over sets.

In the optimization part and constraint part, we express the cost function that has to be optimized, and pose constraints. The primitive constraints, relations, and expressions
Table 7.1: Keywords

<table>
<thead>
<tr>
<th>bseq</th>
<th>bijective</th>
<th>card</th>
</tr>
</thead>
<tbody>
<tr>
<td>diff</td>
<td>forall</td>
<td>in</td>
</tr>
<tr>
<td>int</td>
<td>injective</td>
<td>inter</td>
</tr>
<tr>
<td>max</td>
<td>maximize</td>
<td>min</td>
</tr>
<tr>
<td>minimize</td>
<td>not</td>
<td>ordered</td>
</tr>
<tr>
<td>perm</td>
<td>set</td>
<td>seq</td>
</tr>
<tr>
<td>solve</td>
<td>subject</td>
<td>subset</td>
</tr>
<tr>
<td>sum</td>
<td>surjective</td>
<td>to</td>
</tr>
<tr>
<td>union</td>
<td>var</td>
<td></td>
</tr>
</tbody>
</table>

are the usual ones for (integer) arithmetic, (Boolean) logic, and sets. Powerful aggregation operators such as summation (\texttt{sum}) and the universal quantifier (\texttt{forall}) are also available.

We now present the syntax that we use to declare function problems, which is inspired by the one of OPL\cite{75} but extends it to include function, permutation, and sequence variables and operations on variables of these types.

The basic building blocks are integers (non-terminal \langle Integer \rangle), identifiers (non-terminal \langle Id \rangle), and the keywords of the language (e.g., \texttt{forall}). Integers are sequences of digits, possibly prefixed by a minus sign (\texttt{\textasciitilde}). The keywords are listed in Table 7.1.

The grammar

Function models consist of a set of declarations followed by an instruction:

\[
\langle Model \rangle \rightarrow \{ \langle Declaration \rangle \} \langle Instruction \rangle
\]

Declarations are either input declarations or decision variable declarations:

\[
\langle Declaration \rangle \rightarrow \langle DataDecl \rangle ;
\rightarrow \texttt{var} \langle VarDecl \rangle ;
\]

An input can be declared to be of the primitive types (namely integers and integer sets), as well as relation or function types (declared using the cross-product and function type-constructors). The grammar of data declarations, is as follows:

\[
\langle DataDecl \rangle \rightarrow \langle Id \rangle : \langle Type \rangle
\]

\[
\langle Type \rangle \rightarrow \texttt{int}
\rightarrow \texttt{set(int)}
\rightarrow \langle Id \rangle \times \langle Id \rangle
\rightarrow \langle Id \rangle \rightarrow \langle Id \rangle
\]

Decision variables can be declared using the type constructors \texttt{\textasciitilde\textasciitilde} (designating all partial functions between the two argument sets), \rightarrow (designating all total functions between the two argument sets), \texttt{perm} (designating all permutations of the set argument), \texttt{seq} (design-
nating all sequences of elements drawn from the set argument, the size of the sequences
being the value of the second argument), and \texttt{bseq} (designating all sequences of elements
drawn from the set argument, the size of the sequences being upper-bounded by the value
of the second argument). The grammar of decision variable declarations is:

\begin{align*}
\langle \text{VarDecl} \rangle & \rightarrow \langle \text{Id} \rangle : \langle \text{VarType} \rangle \\
\langle \text{VarType} \rangle & \rightarrow \langle \text{Id} \rangle \rightarrow \text{set}(\text{int}) \\
& \rightarrow \langle \text{Id} \rangle \rightarrow \langle \text{Id} \rangle \\
& \rightarrow \langle \text{Id} \rangle \rightarrow \langle \text{Id} \rangle \\
& \rightarrow \text{perm}(\langle \text{Id} \rangle) \\
& \rightarrow \text{seq}(\langle \text{Id} \rangle, \langle \text{Id} \rangle) \\
& \rightarrow \text{bseq}(\langle \text{Id} \rangle, \langle \text{Id} \rangle)
\end{align*}

Expressions on integers and sets are constructed from constants, input, and decision vari-
ables, using the traditional (aggregate) operators of arithmetic and sets. The operators
\texttt{card}, \texttt{union}, and \texttt{inter} correspond to the cardinality set operation, the union set
operation, and the intersection set operation, respectively. The grammar of expressions
is:

\begin{align*}
\langle \text{Expression} \rangle & \rightarrow \langle \text{UnOp} \rangle \langle \text{Expression} \rangle \\
& \rightarrow \langle \text{Expression} \rangle \langle \text{BinOp} \rangle \langle \text{Expression} \rangle \\
& \rightarrow \langle \text{AggrOp} \rangle \langle \langle \text{Parameter} \rangle^+ \rangle \langle \text{Expression} \rangle \\
& \rightarrow \langle \text{Integer} \rangle \\
& \rightarrow \langle \text{Argument} \rangle \\
& \rightarrow \langle \langle \text{Expression} \rangle \rangle \\
\langle \text{UnOp} \rangle & \rightarrow + \mid - \mid \text{card} \\
\langle \text{BinOp} \rangle & \rightarrow + \mid - \mid \ast \mid \text{union} \mid \text{inter} \\
\langle \text{AggrOp} \rangle & \rightarrow \text{sum} \mid \text{min} \mid \text{max} \\
\langle \text{Argument} \rangle & \rightarrow \langle \text{Object} \rangle \\
& \rightarrow \langle \text{Id} \rangle \langle \text{Deref} \rangle \\
\langle \text{Object} \rangle & \rightarrow \langle \text{Id} \rangle \\
& \rightarrow \langle \langle \text{Id} \rangle^+ \rangle \\
\langle \text{Deref} \rangle & \rightarrow \sim \langle \text{Id} \rangle
\end{align*}

Formulas on integers and sets are constructed from expressions using the traditional op-
erators of arithmetic (\(=\) stands for equality, \(\geq\) stands for greater than or equal, \(\leq\)
stands for less than or equal, \(>\) stands for strictly greater than or equal, \(<\) stands for strictly
less than or equal, and \(<\!\!<\!\!<\) stands for dis-equality), sets (\(\text{in}\) stands for membership,
\(\not\in\) stands for not membership, and \text{subset} stands for subset), and logic (\& stands for
the logic connective \(\wedge\), \(\lor\) stands for the logic connective \(\vee\), \(\Rightarrow\) stands for
the logic connective \(\rightarrow\), and \(\Leftrightarrow\) stands for the logic connective \(\leftrightarrow\)). The grammar of relations is:
(Conjunctions of) constraints on the decision variables are stated using relations (as above), the traditional quantifier $\forall$ of logic, and further constraints on functions. To enforce that a function is surjective, injective, or bijective, we use the keywords $\text{surjective}$, $\text{injective}$, and $\text{bijective}$, respectively. The grammar of constraints is:

$$
\langle \text{Constraint} \rangle \rightarrow \langle \text{Relation} \rangle \\
\rightarrow \forall (\langle \text{Parameter} \rangle^+) \langle \text{Constraint} \rangle \\
\rightarrow \text{bijective}(\langle \text{Id} \rangle) \\
\rightarrow \text{injective}(\langle \text{Id} \rangle) \\
\rightarrow \text{surjective}(\langle \text{Id} \rangle)
$$

Formal parameters are needed for all aggregate operators and quantifiers. They must be within some bounds (a set), and may be required to satisfy some additional condition; the common condition that they be ordered under “<” can be more simply expressed with the $\text{ordered}$ keyword. The grammar of formal parameters is:

$$
\langle \text{Parameter} \rangle \rightarrow [\text{ordered}] \langle \text{Object} \rangle^+ \text{ in } \langle \text{Bounds} \rangle [ : \langle \text{Relation} \rangle ] \\
\langle \text{Bounds} \rangle \rightarrow \langle \text{Argument} \rangle
$$

The instruction posts the constraints of the problem, and states the optional cost function whose value has to be optimized. The grammar of instructions is:

$$
\langle \text{Instruction} \rangle \rightarrow \text{solve} \langle \text{Constraint} \rangle ; \\
\rightarrow \text{minimize} \langle \text{Expression} \rangle \text{ subject to } \langle \text{Constraint} \rangle ; \\
\rightarrow \text{maximize} \langle \text{Expression} \rangle \text{ subject to } \langle \text{Constraint} \rangle ;
$$

The class and precedence of all operators is given in Table 7.2.

### 7.1.2 A modelling tool versus a modelling language

The grammar presented in the previous section allows the description of many CSPs where the objective is to find a function from a given set into another given set, or a fixed (or bounded) length sequence over a given set, or a permutation over a given set. One might be tempted to consider designing a domain specific language for handling function problems rather than modelling them with current CP languages. There are a number of gains in designing a new modelling language:

- We have a language that lifts the level of abstraction beyond coding.
- Models written in $\mathcal{F}$ are easier to write and to maintain (see Chapter 4).
Table 7.2: Operator Precedences

<table>
<thead>
<tr>
<th>Class</th>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>&lt;=&gt;, =&gt;</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>not</td>
<td>4</td>
</tr>
<tr>
<td>arithmetic</td>
<td>=, &gt;=, &lt;=, &lt;, &lt;&gt;</td>
<td>5</td>
</tr>
<tr>
<td>sets</td>
<td>in, not, in, subset</td>
<td>5</td>
</tr>
<tr>
<td>binary</td>
<td>+, -, union</td>
<td>6</td>
</tr>
<tr>
<td>unary</td>
<td>+, -, card</td>
<td>7</td>
</tr>
<tr>
<td>aggregate</td>
<td>sum, min, max</td>
<td>7</td>
</tr>
<tr>
<td>binary</td>
<td>*, inter</td>
<td>8</td>
</tr>
</tbody>
</table>

- Models written in $\mathcal{F}$ can be translated into executable programs (see Chapter 5).

However, such a modelling language would have full impact only when answers to the following difficult questions can be made:

- Starting with an $\mathcal{F}$ model, it is quite often the case that we have more than one corresponding $\mathcal{L}$ model at the CP level (see Chapter 5). Deciding which model is the best is still a major research question (see Chapter 6).

- Having a modelling language where the users are not able to specify a labelling strategy may prohibit them from improving the efficiency. In such cases, the gain in programming time may be neutralized by the loss in solving time. So either we add proper constructs to $\mathcal{F}$ so as to allow the description of a labelling strategy in an $\mathcal{F}$ model or automatically generate $\mathcal{L}$ models with specific labelling strategies. Addressing the former approach goes beyond the scope of this thesis, while the latter approach is still an art.

- Quite often, the translation of an $\mathcal{F}$ model introduces symmetry in the $\mathcal{L}$ models. One example is when the rows (or the columns) of the 2d array in $Fd2$ can be freely permuted. Little work has been done in the direction of detecting and eliminating such symmetries automatically. Again, the gain in programming time, in this case, may be neutralized by the loss in solving time.

- Adding implied constraints may have a significant impact on efficiency [4, 32]. Automatically generating such constraints is a very difficult research problem. Automatically evaluating such implied constraints is even a tougher problem.

- Designing a new modelling language requires either developing a new constraint solver for the $\mathcal{F}$ models or choosing an existing CP language as a target language, where the generated $\mathcal{L}$ models get rewritten into models of the target language. The first approach is not practical and avoids exploiting already existing solvers that are well implemented, while the second makes the $\mathcal{F}$ language highly dependent on the target language. Only models that can be expressed in the target language will be
generated from $\mathcal{F}$ models. For instance, if we pick OPL as a target language, then models that use set variables cannot be considered.

For all these reasons, we believe that the best way to make use of the contributions in this thesis is by designing a practical modelling tool instead. The modelling tool proposes different $\mathcal{L}$ models of function problems expressed in $\mathcal{F}$. The tool will help the users with modelling function problems by speeding up the process of producing initial alternate models that the user may adopt as a starting point. The tool must meet the following requirements:

- The tool must be independent of any current CP language: The output of the tool should rather be descriptions of possible models such as the $\mathcal{L}$ ones used in Chapter 4 and Chapter 5. The output of the model should be viewed as “design patterns” that describe how to model the problem in CP. Those descriptions encode very important modelling decisions: which variables to use, their domains and the constraint formulations. Proper tools can then translate those models written in $\mathcal{L}$ into constraint programs written in a particular CP language. However, we currently perform these translations by hand as they are not part of our tool Fiona. For instance, the $\mathcal{L}$ models of the BACDP presented in Chapter 4 are easily mapped to OPL models in order to run some experiments. This adds a degree of flexibility to the tool because fixing the target language will eliminate some models that may be beneficial, such as the set variable models when either OPL or SICSTUS PROLOG is the chosen target language.

- The study of injection problems in Chapter 6 allowed us to a-priori eliminate certain models of injection problems. The tool must only include those models that survived the theoretical and asymptotic analysis.

- The tool should be modular: it should be easy to add extra functionalities to the tool such as methods that generate and evaluate ICs [5, 32], symmetry-breaking constraints [38, 24], as well as labelling strategies [54].

Based on these requirements we now present an architecture for such a modelling tool.

### 7.2 Architecture of the Fiona modelling tool

The architecture of the Fiona modelling tool is shown in Figure 7.1. First, the ICs adder enforces an extra property on the input $\mathcal{F}$ model. Such properties are presented in Section 5.3 and get rewritten into a set of ICs in the generated $\mathcal{L}$ models. Also, in Section 6.3, we have shown that such implied constraints may be beneficial. Then, both the input $\mathcal{F}$ model and the $\mathcal{F}$ model with an extra constraint enforcing a certain property serve as an input to the model generator component. Different alternate $\mathcal{L}$ models are generated as an output from the model generator component.

Starting from a function problem description expressed in $\mathcal{F}$, a model generator produces a set of alternate $\mathcal{L}$ models, with some explanations added as comments in each of these models. The detailed architecture of the model generator is shown in Figure 7.2.

The sub-component $Fd1$ generates different models along with their explanations based on the representation $Fd1$ of function variables. The sub-component $Fd2$ uses the 2d 0/1 array in the representation $Fd2$ to derive some of the generated models, while $S$ uses
set variables. The generation of such models is achieved through the use of the results presented in section 5.1. Generating combined models is the responsibility of the sub-
components \( Fd1+Fd2 \), \( Fd1+S \), and \( Fd2+S \), which generate combined models as described in Section 5.2 by employing the heuristics \( H_{\text{arithmetic}} \), \( H_{\text{inverse}} \), \( H_{\text{range}} \), and \( H_{\text{membership}} \) to choose between different constraint formulations, and adding the appropriate channelling constraints. The sub-component \( Fd1+Fd2 \) generates integrated/hybrid models that combine the representation in \( Fd1 \) with the one in \( Fd2 \). The sub-component \( Fd1+S \) generates integrated models that combine the representation in \( Fd1 \) with the one in \( S \). The sub-component \( Fd2+S \) generates integrated/hybrid models that combine the representation in \( Fd2 \) with the one in \( S \). The model generator can be seen as a non-deterministic \( \mathcal{F} \)-to-\( \mathcal{L} \) compiler. The architecture of such a compiler is composed of four parts. First, the decomposer separates an \( \mathcal{F} \) model into its declaration, optimization, and constraint parts. Next, the \( \mathcal{F} \)-to-\( \mathcal{L} \) declaration converter rewrites all \( \mathcal{F} \) declarations into \( \mathcal{L} \) declarations, and possibly into some \( \mathcal{L} \) constraints. Also, the \( \mathcal{F} \)-to-\( \mathcal{L} \) constraint converter rewrites the \( \mathcal{F} \) optimization and constraint parts into an \( \mathcal{L} \) optimization part and more \( \mathcal{L} \) constraints, using the declaration part. Finally, the composer assembles the generated \( \mathcal{L} \) models by suitably concatenating the obtained \( \mathcal{L} \) statements.

The decomposer and composer modules are trivial, and are not discussed here. Next, we will show that the converter modules are easily implemented by a set of \( \mathcal{F} \)-to-\( \mathcal{L} \) rewrite rules.

### 7.2.1 \( \mathcal{F} \)-to-\( \mathcal{L} \) rewrite rules

We use conditional rewrite rules, here written as follows:

\[
L \Rightarrow R \mid C
\]

meaning that, if condition \( C \) holds, then expression \( L \) is rewritten into \( R \).

As a running example, we show how each line of an \( \mathcal{F} \) model of the BACDP (see Chapter 4 for a description), shown in Figure 7.3, is compiled into some line(s) of the \( \mathcal{L} \) models in Figure 7.4, Figure 7.5, Figure 7.6, Figure 7.7, Figure 7.8, and Figure 7.9. Note that line annotations are added for future reference. Also, note that all \( \mathcal{L} \) models are automatically generated by the current Fiona prototype system\(^1\).

#### \( \mathcal{F} \)-to-\( \mathcal{L} \) declaration converter

All input declarations become identical \( \mathcal{L} \) input declarations. For instance, lines a–i are input declarations, and hence will be present in any output \( \mathcal{L} \) model, but we do not show them (here) in the generated models.

The declarations of \( \mathcal{F} \) that involve types not supported by \( \mathcal{L} \) (namely functions, permutations, and sequences) are rewritten into \( \mathcal{L} \) declarations, and possibly into some \( \mathcal{L} \) constraints.

For total function variable declarations, we have multiple rules, which are based on the results presented in Chapter 5.

The rule:

\[
\text{var } F : V \rightarrow W; \quad \Rightarrow \quad \text{var } \text{int } F.d1[V] \text{ in } W; \quad \% \quad F.d1[i] = j \iff \langle i, j \rangle \text{ belongs to } F \\
| \quad V \text{ and } W \text{ are given sets}
\]

\(^1\)The Fiona prototype system is available upon request.
refines the function variable $F$ as a 1d array $F_{d1}$. For instance, line j in Figure 7.3 gets rewritten into lines 1-2 in Figure 7.4. Note that, we also add some comments to the generated $L$ statement relating the variables at the $L$ level of abstraction with the function variable.

The rule:

\[
\text{var } F: V\rightarrow W; \quad \Rightarrow \quad \text{var int } F_{d2}[V,W] \text{ in } 0..1; \\
\frac{\% F_{d2}[i,j]=1 \text{ iff } <i,j> \text{ belongs to } F}{\% \text{ every } i \text{ in } V \text{ gets assigned exactly 1 element in } W} \\
\frac{\text{forall}(i \text{ in } V) \text{ sum}(j \text{ in } W) F_{d2}[i,j] = 1;}{V \text{ and } W \text{ are given sets}}
\]

refines the function variable $F$ as a 2d Boolean array $F_{d2}$ and adds extra row-sum constraints. For instance, line j in Figure 7.3 gets rewritten into lines 1-2 and lines 7-8 in Figure 7.5. Note also that some explanation of the extra constraints is encoded as a comment in the rewrite rule.

The rule:

\[
\text{var } F: V\rightarrow W; \quad \Rightarrow \quad \text{var } F_{S}[W] \subset V; \\
\frac{\% i \text{ in } F_{S}[j] \text{ iff } <i,j> \text{ belongs to } F}{\% \text{ every } i \text{ in } V \text{ gets assigned exactly 1 element in } W} \\
\frac{\text{union all}(j \text{ in } W) F_{S}[j] = V;}{\text{forall}(i \text{ in } V) \text{forall}(j \text{ in } W) i<>j=>F_{S}[i] \text{ inter } F_{S}[j]={};} \\
\frac{V \text{ and } W \text{ are given sets}{}}{}
\]
represents the function variable \( F \) using set variables and extra constraints. For instance, line \( j \) in Figure 7.3 gets rewritten into lines 1-2 and lines 11-13 in Figure 7.6.

The rule:

\[
\text{var } F: V\rightarrow W; \\
\Rightarrow \text{var int } F_{d1}[V] \text{ in } W; \\
\quad \% F_{d1}[i]=j \text{ iff } <i,j> \text{ belongs to } F \\
\quad \text{var int } F_{d2}[V,W] \text{ in } 0\ldots 1; \\
\quad \% F_{d2}[i,j]=1 \text{ iff } <i,j> \text{ belongs to } F \\
\quad \% \text{ every } i \text{ in } V \text{ gets assigned exactly 1 element in } W \\
\quad \text{forall}(i \in V) \text{ sum}(j \in W) F_{d2}[i,j] = 1; \\
\quad \% \text{ Channelling constraints} \\
\quad \text{forall}(i \in V, j \in W) F_{d1}[i]=j \iff F_{d2}[i,j] = 1; \\
\quad | \quad V \text{ and } W \text{ are given sets} \\
\]

integrates the 1d array \( F_{d1} \) with the 2d Boolean array \( F_{d2} \). The two sets of variables are connected through channelling constraints. For instance, line \( j \) in Figure 7.3 gets rewritten into lines 1-4 and lines 9-12 in Figure 7.7.

The rule:

\[
\text{var } F: V\rightarrow W; \\
\Rightarrow \text{var int } F_{d1}[V] \text{ in } W; \\
\quad \% F_{d1}[i]=j \text{ iff } <i,j> \text{ belongs to } F \\
\quad \text{var } F_{S}[W] \text{ subset } V; \\
\quad \% i \in F_{S}[j] \text{ iff } <i,j> \text{ belongs to } F \\
\quad \% \text{ every } i \text{ in } V \text{ gets assigned exactly 1 element in } W \\
\quad \text{union all}(j \in W) F_{S}[j] = V; \\
\quad \text{forall}(i \in W) \text{ forall}(j \in W) i<>j=>F_{S}[i] \text{ inter } F_{S}[j]=\{}; \\
\quad \% \text{ Channelling constraints} \\
\quad \text{forall}(i \in V, j \in W) F_{d1}[i]=j \iff i \in F_{S}[j]; \\
\quad | \quad V \text{ and } W \text{ are given sets} \\
\]

integrates the 1d array \( F_{d1} \) with the set variables representation \( F_{S} \). The two sets of variables are connected through channelling constraints. For instance, line \( j \) in Figure 7.3 gets rewritten into lines 1-4 and lines 9-13 in Figure 7.8.

The rule:

\[
\text{var } F: V\rightarrow W; \\
\Rightarrow \text{var int } F_{d2}[V,W] \text{ in } 0..1; \\
\quad \% F_{d2}[i,j]=1 \text{ iff } <i,j> \text{ belongs to } F \\
\quad \text{var } F_{S}[W] \text{ subset } V; \\
\quad \% i \in F_{S}[j] \text{ iff } <i,j> \text{ belongs to } F \\
\quad \% \text{ every } i \text{ in } V \text{ gets assigned exactly 1 element in } W \\
\quad \text{forall}(i \in V) \text{ sum}(j \in W) F_{d2}[i,j] = 1; \\
\quad \% \text{ every } i \text{ in } V \text{ gets assigned exactly 1 element in } W \\
\quad \text{union all}(j \in W) F_{S}[j] = V; \\
\quad \text{forall}(i \in W) \text{ forall}(j \in W) i<>j=>F_{S}[i] \text{ inter } F_{S}[j]=\{}; \\
\quad \% \text{ Channelling constraints} \\
\quad \text{forall}(i \in V, j \in W) F_{d2}[i,j]=1 \iff i \in F_{S}[j]; \\
\quad | \quad V \text{ and } W \text{ are given sets} \\
\]
integrates the 2d Boolean array \( F_{d2} \) with the set variables representation \( F_{S} \). The two sets of variables are connected through channelling constraints. For instance, line \( j \) in Figure 7.3 gets rewritten into lines 1-4 and lines 9-15 in Figure 7.9.

Finally, the rule:

\[
\text{var } F : V \rightarrow W; \\
\Rightarrow \text{var int } F_{d1}[V] \text{ in } 0..\text{maxint}; \\
\% F_{d1}[i]=j \text{ iff } <i,j> \text{ belongs to } F \\
\mid V \text{ is a given set and } W \text{ is int}
\]

takes care of the special case when the set \( W \) is not given. Then, we represent the function variable using only one representation, which is the 1d array \( F_{d1} \). For instance, line \( k \) in Figure 7.3 gets rewritten into lines 3-4 in Figures 7.4, 7.5 and 7.6 and into lines 5-6 in Figures 7.7, 7.8 and 7.9.

\( F \)-to-\( L \) constraint converter

The expressions and constraints in an \( F \) model get rewritten into expressions, constraints, and possibly \( L \) variables depending on the representation choice for the \( F \) variables. The results presented in Chapter 5 can be easily encoded using \( F \)-to-\( L \) rewrite rules.

The objective function in the \( F \) model in Figure 7.3 minimizes the range of function \( \text{Load} \). This is rewritten in the same way in all models because \( \text{Load} \) is represented as \( \text{Load}_{d1} \) in all generated \( L \) models. The following rewrite rule achieves that:

\[
\text{max}(\text{range}(F)) \\
\Rightarrow \text{var int maxrange in } 0..\text{maxint}; \\
\text{maxrange} \\
\max(\text{maxrange}); \\
\mid F \text{ is represented as } F_{d1}
\]
1: var int Cur_d2[courses,periods] in 0..1;
2: % Cur_d2[i,j]=1 iff <i,j> belongs to Cur
3: var int Load_d1[periods] in 0..maxint;
4: % Load_d1[i]=j iff <i,j> belongs to Load
5: var int maxrange in 0..maxint;
6: minimize
7: maxrange subject to {
8: % every i in courses gets assigned exactly 1 element in periods
9: forall(i in courses) sum(j in periods) Cur_d2[i,j] = 1;
10: forall(p in periods) minload <= Load_d1[p];
11: forall(p in periods) maxload >= Load_d1[p];
12: forall(p in periods) minnbcourses <= sum(i in courses) Cur_d2[i,p];
13: forall(p in periods) maxnbcourses >= sum(i in courses) Cur_d2[i,p];
14: forall(<c1,c2> in prereq) sum(j in periods) Cur_d2[c1,j] * j < sum(j in periods) Cur_d2[c2,j] * j;
15: forall(p in periods) Load_d1[p] = weighted-cardinality(Cur_d2[p],crefun);
16: max(Load_d1,maxrange)};

Figure 7.5: An \( \mathcal{L} \) model of the BACDP based on \( Fd^2 \)

1: var Cur_S[periods] subset courses;
2: % i in Cur_S[j] iff <i,j> belongs to Cur
3: var int Load_d1[periods] in 0..maxint;
4: % Load_d1[i]=j iff <i,j> belongs to Load
5: var int Cur_B[courses,periods] in 0..1;
6: % redundant Boolean representation for Cur
7: var int maxrange in 0..maxint;
8: minimize
9: maxrange subject to {
10: % Channelling constraints
11: forall(i in courses, j in periods) i in Cur_S[j] <=> Cur_B[i,j] = 1;
12: forall(i in courses) sum(j in periods) Cur_S[j] = courses;
13: forall(p in periods) minnbcourses <= card(Cur_S[p]);
14: forall(p in periods) maxnbcourses >= card(Cur_S[p]);
15: forall(<c1,c2> in prereq) sum(j in periods) Cur_B[c1,j] * j < sum(j in periods) Cur_B[c2,j] * j;
16: max(Load_d1,maxrange)};

Figure 7.6: An \( \mathcal{L} \) model of the BACDP based on \( S \)

So, line 1 get rewritten into lines 7,8,17 in Figure 7.4, lines 5,6,16 in Figure 7.5, lines 7,8,21 in Figure 7.6, lines 7,8,25 in Figure 7.7, lines 7,8,24 in Figure 7.8, and lines 7,8,30 in Figure 7.9.
1: var int Cur_d1[courses] in periods;  
2: % Cur_d1[i]=j iff <i,j> belongs to Cur  
3: var int Cur_d2[courses,periods] in 0..1;  
4: % Cur_d2[i,j]=1 iff <i,j> belongs to Cur  
5: var int Load_d1[periods] in 0..maxint;  
6: % Load_d1[i]=j iff <i,j> belongs to Load  
7: var int maxrange in 0..maxint;  
8: minimize maxrange subject to {  
9: % every i in courses gets assigned exactly 1 element in periods  
10: forall(i in courses) sum(j in periods) Cur_d2[i,j] = 1;  
11: % Channelling constraints  
12: forall(i in courses, j in periods) Cur_d1[i] = j <==> Cur_d2[i,j] = 1;  
13: forall(p in periods) Load_d1[p] = sum(c in courses) Cur_d2[c,p]*crefun(c);  
14: forall(p in periods) minload <= Load_d1[p];  
15: forall(p in periods) maxload >= Load_d1[p];  
16: forall(p in periods) atleast(p,Cur_d1,minnbcourses);  
17: /* Use global constraint when a CP solver is used */  
18: /* When a CP+ILP solver is used also use this linear formulation:  
19: minnbcourses <= sum(i in courses) Cur_d2[i,p] */  
20: forall(p in periods) atmost(p,Cur_d1,maxnbcourses);  
21: /* Use global constraint when a CP solver is used */  
22: /* When a CP+ILP solver is used also use this linear formulation:  
23: maxnbcourses >= sum(i in courses) Cur_d2[i,p] */  
24: forall(<c1,c2> in prereq) Cur_d1[c1] < Cur_d1[c2];  
25: max(Load_d1, maxrange});

Figure 7.7: An $L$ model of the BACDP based on $Fd1 + Fd2$

Note that we have multiple rewrite rules when the range of a function is involved in some expression, depending on the context. For each context, a different rewrite rule is developed. For example, if we instead had the objective function $\text{card}($range$(F))$, then the following rule would have been employed instead:

\[
\text{card}($\text{range}(F)) \\
\Rightarrow \quad \text{var int range}_F[W] \text{ in } 0..1;  
\text{sum}(j \text{ in } W) \text{ range}_F[j]  
\%\text{linking constraints}  
\text{forall}(i \text{ in } V) \text{ forall}(j \text{ in } W) \text{ F}_d1[i]=j \Rightarrow \text{ range}_F[j]=1;  
\text{Total function } F \text{ from } V \text{ into } W \text{ is represented as } F_d1
\]

As for function application we again have multiple rewrite rules. By applying this rule:

\[
F(i)  
\Rightarrow F_d1[i]  
\text{Total function } F \text{ is represented as } F_d1
\]

all occurrences of $\text{Load}(i)$ in the $F$ model in Figure 7.3 are rewritten into $\text{Load}_d1[i]$ in all generated $L$ models.

The rule:

117
1: var int Cur_d1[courses] in periods;
2: % Cur_d1[i]=j iff <i,j> belongs to Cur
3: var Cur_S[periods] subset courses;
4: % i in Cur_S[j] iff <i,j> belongs to Cur
5: var int Load_d1[periods] in 0..maxint;
6: % Load_d1[i]=j iff <i,j> belongs to Load
7: var int maxrange in 0..maxint;
8: maximize
9: maxrange subject to {
10: union all(j in periods) Cur_S[j] = courses;
11: forall(i in periods) forall(j in periods)i<>j=>Cur_S[i] inter Cur_S[j]={};
12: Channelling constraints
13: forall(i in courses, j in periods) Cur_d1[i]=j <=> i in Cur_S[j];
14: forall(p in periods) Load_d1[p] = weighted-cardinality(Cur_S[p],crefun);
15: forall(p in periods) minload <= Load_d1[p];
16: forall(p in periods) maxload >= Load_d1[p];
17: forall(p in periods) atleast(p,Cur_1d,minnbcourses);
18: /* You may also use this formulation:
19: minnbcourses<= card(Cur_S[p])/ /
20: forall(p in periods) atmost(p,Cur_d1,maxnbcourses);
21: */ You may also use this formulation:
22: maxnbcourses>= card(Cur_S[p])/ /
23: forall(<c1,c2> in prereq) Cur_d1[c1] < Cur_d1[c2];
24: max(Load_d1, maxrange)};

Figure 7.8: An L model of the BACDP based on Fd1 + S

\[ F(i) \Rightarrow \text{\textbf{sum}}(j \text{ in } W) F_{d2}[i,j]*j \]
| Total function \( F \) from \( V \) into \( W \) is represented as \( F_{d2} \)

The rule:

\[ F(i) \Rightarrow \% \text{ redundant Boolean representation for } F \]
\[ \text{var int } F_B[V,W] \text{ in } 0..1; \]
\[ \% \text{ Channeling constraints} \]
\[ \text{forall}(i \text{ in } V, j \text{ in } W) \text{ i in } F_S[j] \leftrightarrow F_B[i,j] = 1; \]
\[ \text{sum(j in W) } F_B[i,j] * j \]
| Total function \( F \) from \( V \) into \( W \) is represented as \( F_S \)

rewrites line \( r \) in Figure 7.3 into line 15 in Figure 7.5.

When we consider an integrated/hybrid model, we have a choice to make between the two formulations. However, we incorporate the heuristic \( H_{\text{arithmetic}} \) presented in Chapter 5 into the rewrite rules as follows.

The rule:

\[ F(i) \Rightarrow F_{d1}[i] \]
| Total function \( F \) is represented as \( F_{d1} \) and \( F_{d2} \)
chooses the formulation on $F_{d1}$, and hence rewrites line $r$ in Figure 7.3 into line 24 in Figure 7.7.

Similar to the previous rule, the formulation on $F_{d1}$ for line $r$ in Figure 7.3 is rewritten into line 23 in Figure 7.8 by applying this rule:

$$F(i) \Rightarrow F_{d1}[i]$$

| Total function $F$ is represented as $F_{d1}$ and $F_S$

The rule:

$$F(i) \Rightarrow \sum(j \in W) F_{2}[i,j]*j$$

| Total function $F$ from $V$ into $W$ is represented as $F_{d2}$ and $F_S$

$F(i)$ appears in a linear constraint.
chooses the constraint formulation on $F_{d2}$ and rewrites line $r$ in Figure 7.3 into lines 28-29 in Figure 7.9.

This rule, however, is not applicable in our case and hence will not be fired.

\[
F(i) \Rightarrow \%	ext{ redundant Boolean representation for } F \\
\text{var int } F_B[V,W] \text{ in } 0..1; \\
\%	ext{ Channelling constraints} \\
\text{forall}(i \in V, j \in W) i \in F.S[j] \iff F_B[i,j] = 1; \\
\text{sum}(j \in W) F_B[i,j] \ast j \\
| \text{Total function } F \text{ from } V \text{ into } W \text{ is represented as } F_{d2} \text{ and } F_S \\
F(i) \text{ appears in a non-linear constraint.}
\]

Our last rewrite rules are concerned with expressions and constraints that involve the inverse function application operation. We distinguish between two contexts that appear in the example. Assume we have a total function $F$ from $V$ into $W$.

In the first case, we have a sum expression that iterates over the elements of $F_{\sim j}$ (i.e. $F^{-1}(j)$). This expression appear in an arithmetic constraint, which results in weighted capacity constraints. We have multiple rewrite rules.

The rule:

\[
\text{sum}(i \in F_{\sim j}) Q(i) \\
\Rightarrow \%	ext{ redundant Boolean representation for } F \\
\text{var int } F_B[V,W] \text{ in } 0..1; \\
\%	ext{ Channelling constraints} \\
\text{forall}(i \in V, j \in W) F.d1[i]=j \iff F_B[i,j] = 1; \\
\text{sum}(i \in V) F_B[i,j] \ast Q(i) \\
| \text{Total function } F \text{ from } V \text{ into } W \text{ is represented as } F_{d1}
\]

rewrites parts of line $m$ in Figure 7.3 into lines 6,7,9-11 in Figure 7.4.

The rule:

\[
\text{sum}(i \in F_{\sim j}) Q(i) \\
\Rightarrow \text{sum}(i \in V) F_{d2}[i,j] \ast Q(i) \\
| \text{Total function } F \text{ from } V \text{ into } W \text{ is represented as } F_{d2}
\]

rewrites parts of line $m$ in Figure 7.3 into parts of line 9 in Figure 7.5.

The rule:

\[
\text{sum}(i \in F_{\sim j}) Q(i) \\
\Rightarrow \text{weighted-cardinality}(F.S[j],Q) \\
| \text{Total function } F \text{ from } V \text{ into } W \text{ is represented as } F_S
\]

This rule rewrites some parts of line $m$ in Figure 7.3:

\[
\text{sum}(c \text{ in Cur.~}^{\sim}p)\text{crefun}(c) \\
\text{into parts of line 14 in Figure 7.6:}
\]

\[
\text{weighted-cardinality}(\text{Cur}_S[p],\text{crefun})
\]

As for the integrated/hybrid models, we incorporate the heuristic $H_{inverse}$ presented in Chapter 5 into the following rewrite rules in order to pick a suitable constraint formulation.

The rule:
sum(i in F.∼j) Q(i) ⇒ sum(i in V) F.d2[i,j] * Q(i)  
| Total function F from V into W is represented as F.d1 and F.d2

rewrites parts of line m in Figure 7.3 into parts of line 13 in Figure 7.7 by choosing the constraint formulation on F.d2.

The rule:

sum(i in F.∼j) Q(i) ⇒ weighted-cardinality(F.S[j],Q)  
| Total function F from V into W is represented as F.d1 and F.S

chooses the constraint formulation on F.S and rewrites parts of line m in Figure 7.3 into parts of line 14 in Figure 7.8.

When both the representations F.d2 and F.S are used, the heuristic H_{inverse} cannot distinguish between the formulations and this is reflected in the rewrite rule by providing the two formulation, but one of them is inserted as a comment. This rule rewrites parts of line m in Figure 7.3 into lines 16-17 in Figure 7.9:

sum(i in F.∼j) Q(i) ⇒ sum(i in V) F.d2[i,j] * Q(i)  
%Or: weighted-cardinality(F.S[j],Q)  
| Total function F from V into W is represented as F.d2 and F.S

In the second case, the cardinality of F.∼j is restricted in lines p-q in Figure 7.3, which results in capacity constraints. We again have multiple rewrite rules.

The rule:

card(F.∼j) ⇒ atleast(j,F.d1,E)  
| Total function F from V into W is represented as F.d1  
card(F.∼j) appears in the context: E <= card(F.∼j)  
where E is any given integer input

rewrites parts of line p in Figure 7.3 into parts of line 14 in Figure 7.4.

A similar rule:

card(F.∼j) ⇒ atmost(j,F.d1,E)  
| Total function F from V into W is represented as F.d1  
card(F.∼j) appears in the context: E >= card(F.∼j)  
where E is any given integer input

rewrites parts of line q in Figure 7.3 into parts of line 15 in Figure 7.4.

The rule:

card(F.∼j) ⇒ sum(i in V) F.d2[i,j]  
| Total function F from V into W is represented as F.d2

This rule rewrites parts of line p-q in Figure 7.3 into parts of lines 12-13 in Figure 7.5.

The rule:
card(F.∼j) ⇒ card(F_S[j])

| Total function F from V into W is represented as F_S

rewrites parts of line p-q in Figure 7.3 into parts of line 17-18 in Figure 7.6.

As for integrated/hybrid models, we have the following rules that incorporate the heuristic \( \mathcal{H}_{\text{inversed}} \).

The rule:

\[
\text{card}(F.\sim j) \Rightarrow \text{atleast}(j,F_{d1},E)
\]

/* Use global constraint when a CP solver is used */
/* When a CP+ILP solver is used also use this linear formulation:
E <= \text{sum}(i \in V) F_{d2}[i,j] */
| Total function F from V into W is represented as F_{d1} and F_{d2}
| card(F.∼j) appears in the context: E <= card(F.∼j)
where E is any given integer input

rewrites parts of line p in Figure 7.3 into parts of lines 16-19 in Figure 7.7. A similar rule rewrites parts of line q in Figure 7.3 into parts of lines 20-23 in Figure 7.7.

The rule:

\[
\text{card}(F.\sim j) \Rightarrow \text{atleast}(j,F_{d1},E)
\]

/* You may also use this formulation:
E <= \text{card}(F_S[j]) */
| Total function F from V into W is represented as F_{d1} and F_S
| card(F.∼j) appears in the context: E <= card(F.∼j)
where E is any given integer input

rewrites parts of line p in Figure 7.3 into parts of lines 17-19 in Figure 7.8. A similar rule rewrites parts of line q in Figure 7.3 into parts of lines 20-22 in Figure 7.8.

Finally, this rule:

\[
\text{card}(F.\sim j) \Rightarrow \text{card}(F_S[j])
\]

/* Use global constraint when a CP solver is used */
/* When a CP+ILP solver is used also use this linear formulation:
\text{sum}(i \in V) F_{d2}[i,j] */
| Total function F from V into W is represented as F_{d2} and F_S

rewrites parts of line p in Figure 7.3 into parts of lines 20-23 in Figure 7.9. A similar rule rewrites parts of line q in Figure 7.3 into parts of lines 24-27 in Figure 7.9.

7.3 Evaluation of the tool

Our aim in this section is to evaluate our hypothesis, which is two-fold:

1. \( \mathcal{F} \) provides a high-level of abstraction that allows us to easily write, maintain, and update models that cover a wide range of applications from different domains.
2. The generated \( \mathcal{L} \) models from an \( \mathcal{F} \) model are reasonably efficient, but there is no best \( \mathcal{L} \) model.

Possible criteria that one might use to evaluate the first part of the hypothesis can be:

- **Maintenance**: how easily can we maintain an \( \mathcal{F} \) model? We consider an \( \mathcal{F} \) model of a given problem, change the problem slightly and measure the difficulty of changing the \( \mathcal{F} \) model to accommodate the change. In Chapter 4, we argued that it is easier to update and maintain models expressed in \( \mathcal{F} \). With the graph coloring problem, we showed that it is easy to update the \( \mathcal{F} \) model for the satisfaction problem to cover the optimization one, while with the warehouse location problem, we showed that it is easy to maintain the \( \mathcal{F} \) model when we have a minor change in the requirements.

- **Scope**: how wide is the scope of the \( \mathcal{F} \) language? We select 10 problems from CSPLIB covering all the available domains, namely, the scheduling domain, the design and configuration domain, the bin packing, the frequency assignment domain, the combinatorial mathematics domain, the games and puzzles domain, and the bioinformatics domain. Then, we attempt to model these problems in \( \mathcal{F} \). The ratio of the problems for which we have an \( \mathcal{F} \) model would give us an idea about the scope of the \( \mathcal{F} \) language.

- **Ease of use**: how easily can we write an \( \mathcal{F} \) model? Ideally, one has to perform a usability study. For instance, one might select a number of users and split them into two groups A and B. Group A is supposed to use \( \mathcal{F} \) while group B is supposed to use another CP language, say \( \text{opl} \). The same problems are given to the two groups and the group that can model the most number of problems in a fixed period of time will give us clues on the ease of use of \( \mathcal{F} \) compared to \( \text{opl} \). However, due to the practical difficulties involved, we consider performing this task in our future work.

As for assessing the second part of the hypothesis, we will use the following criteria:

- **Analysis of the modelling decisions made by the tool**: The quality of the generated \( \mathcal{L} \) models depends on the modelling decisions we make on how to represent the variables and on how to state the problem constraints on these variables.

- **Empirical evaluation of the generated \( \mathcal{L} \) models**: We will take an empirical approach towards establishing that there is no best \( \mathcal{L} \) model. We will compare generated \( \mathcal{L} \) models of different problems on different instances and show that there is no best \( \mathcal{L} \) model.

- **Comparison with hand-crafted models**: To further assess the quality of the generated \( \mathcal{L} \) models, a comparison of some hand-crafted models of some problems with the generated \( \mathcal{L} \) models is necessary.

### 7.3.1 Scope of the \( \mathcal{F} \) language

Now, we will try to assess the scope of the \( \mathcal{F} \) language by going through a number of problems from different domains and show an \( \mathcal{F} \) formulation for each of them, if possible.

**Bin packing problems**

We consider the Schur’s lemma problem (Prob015) as our example of a bin packing problem.
balls: \{\text{int}\};
buses: \{\text{int}\};
var Schur: \text{balls} \rightarrow \text{boxes};
solve { 
forall(i,j,k \in \text{balls}: i+j=k) 
\neg(\text{Schur}(i) = \text{Schur}(j) = \text{Schur}(k));
}

Figure 7.10: An $\mathcal{F}$ model of the Schur's lemma problem

maintcost : \text{int};
stores: \{\text{int}\};
warehouses : \{\text{int}\};
capacity : \text{warehouses} \rightarrow \text{int};
supplycost: \text{stores} \times \text{warehouses} \rightarrow \text{int};
var SUPPLIER : \text{stores} \rightarrow \text{warehouses};
minimize 
\sum_{\langle i,j \rangle \in \text{SUPPLIER}} \text{supplycost}(<i,j>)+
\text{card(range(SUPPLIER))}\times\text{maintcost}
subject to{
forall (j \in \text{warehouses}) \text{card(SUPPLIER.~(j))} \leq \text{capacity}(j));

Figure 7.11: An $\mathcal{F}$ model of the WLP

\textbf{Schur's lemma problem} \hspace{1em} The problem is to put $n$ balls labelled \{1, \ldots, n\} into 3 boxes so that for any triple of balls $\langle x, y, z \rangle$ with $x + y = z$, not all are in the same box.

From the problem description, one might think of the set of balls and the set of boxes. When the desired output is captured by a total function from the set of balls into the set of boxes, the only problem constraint can be formulated in a straightforward manner following the problem description. An $\mathcal{F}$ model of the Schur problem is shown in Figure 7.10. Please note the close correspondence between the $\mathcal{F}$ model and the problem description in natural language.

\textbf{Design and configuration}

We consider two problems. We just show an $\mathcal{F}$ model of the WLP (Prob034) already presented and discussed in Chapter 4, and consider the Balanced Incomplete Block Design problem (Prob028).

\textbf{Warehouse location problem.} \hspace{1em} The $\mathcal{F}$ model of the WLP shown in Figure 7.11 is just the model in Figure 4.8 presented in Chapter 4 but written in the $\mathcal{F}$ syntax.

\textbf{Balanced incomplete block design problem.} \hspace{1em} Balanced Incomplete Block Design (BIBD) generation is a standard combinatorial problem from design theory, originally used in the design of statistical experiments but since finding other applications such as cryptography.

A BIBD is defined as an arrangement of $v$ distinct objects into $b$ blocks such that each block contains exactly $k$ distinct objects, each object occurs in exactly $r$ different blocks, and every two distinct objects occur together in exactly $l$ blocks. Another way of defining
A BIBD is in terms of its incidence matrix, which is a $v \times b$ binary matrix with exactly $r$ ones per row, $k$ ones per column, and with a scalar product of $l$ between any pair of distinct rows. A BIBD is therefore specified by its parameters $(v, b, r, k, l)$.

From the problem description, one might think of the set of objects and the set of blocks. However, the relationship between these two sets can be captured by a function from the set of objects times the set of blocks into the set $\{0, 1\}$. However, we restrict ourselves to functions where the target set or the source set are not sets of tuples. Furthermore, a relation between these two sets best represents the desired output in the BIBD problem.

**Scheduling**

In addition to the BACDP (Prob030) presented in Chapter 4 and used as a running example in the previous section, we consider another scheduling problem: a Bus Driver Scheduling problem (Prob022).

**Bus driver scheduling problem** The Bus Driver Scheduling problem consists of a given set of tasks (pieces of work) to cover and a large set of possible shifts, where each shift covers a subset of the tasks and has an associated cost. We must select a subset of possible shifts that covers each piece of work once and only once: this is called a partition. Further, in the driver scheduling (unlike air crew scheduling) the main aim is to reduce the number of shifts used in the solution partition and the total cost of the partition is secondary. To simplify the problem the cost of each shift is the same. This means that the goal is to minimize the number of shifts.

An $\mathcal{F}$ model of the bus driver scheduling problem is shown in Figure 7.12. The inputs are captured by the set of pieces of work $\text{pieces}$, the set of shift indexes $\text{shifts}$, and a total function $\text{coverage}$ that returns the set of pieces covered by each shift. The desired output is captured by a total function $\text{Schedule}$ from $\text{pieces}$ into $\text{shifts}$. To express that a piece $i$ must not be assigned a shift $j$ that does not cover it, we use a dis-equality constraint whenever the coverage of $j$ does not include $i$. Furthermore, in order to make sure that each piece of work is covered once and only once, we enforce that whenever a shift is in the range of the function (i.e., a shift is used), then the cardinality of the set of pieces that have image that particular shift must be equal to the number of pieces that that shift covers. Finally, we state the objective function as a minimization of the cardinality of the range of $\text{Schedule}$.

**Frequency assignment**

We consider the Golomb Rulers problem (Prob006) as an example of a frequency assignment problem.

**Golomb rulers problem** Golomb Rulers problems have many practical applications including sensor placements for x-ray crystallography and radio astronomy. A Golomb Ruler may be defined as a set of $m$ integers $0 = a_1 < a_2 < \ldots < a_m$ such that the $m(m-1)/2$ differences $a_j - a_i$, $1 \leq i < j \leq m$ are distinct. Such a ruler is said to contain $m$ marks and is of length $a_m$. The objective is to find optimal (minimum length) or near optimal rulers.
pieces: {int};
shifts: {int};
coverage: shifts->{int};
var Schedule: pieces->shifts;
minimize
   card(range(Schedule))
subject to {
   forall(j in shifts)
      forall(i in pieces: i not in coverage(j))
         Schedule(i) <> j;
  forall(j in range(schedule)) card(Schedule."(j)) = card(coverage(j));
}

Figure 7.12: An $F$ model of the bus driver scheduling problem

ticks: {int}; % ticks={1,..,m}
distances: {int}; % distance has m(m-1)/2 values
pair:distance->ticks x ticks;
% every distance has an image an ordered pair of ticks
var Tick-assign: ticks->int;
var Distance-assign: distances->int;
minimize
   max(range(Tick-assign))
subject to {
   Tick-assign(1)=0;
   forall(i in ticks: i<m)
      Tick-assign(i) < Tick-assign(i+1);
injective(Tick-assign);
injective(Distance-assign);
   forall(<a,<b,c>> in pair)
      Distance-assign(a) = Tick-assign(b) - Tick-assign(c));

Figure 7.13: An $F$ model of the golomb ruler problem

A possible $F$ model of the Golomb Ruler problem is shown in Figure 7.13. The inputs are the set of ticks $ticks$, the set of distance indexes $distances$, and a function $pair$ mapping each distance index to an ordered pair of ticks. The outputs are captured by two injections: $Tick-assign$ assigns distinct integers to any two distinct ticks and $Distance-assign$ maps any two distinct distances to distinct integers, which captures parts of the problem constraints. We also make sure that the first tick is assigned to 0 and use strict inequalities to enforce an ordering on the images of the ticks. The last constraint states that each distance represents a difference between two ticks. Finally, we guarantee that our ruler is optimal by minimizing the maximum element in the range of $Tick-assign$.

Combinatorial mathematics problems

We consider two problems as our examples of combinatorial mathematics problems: the All-intervals Series problem (Prob007) and the Langford’s problem (Prob024).
All-interval series problem

Given the twelve standard pitch-classes (c, c#, d, ...), represented by numbers 0,1,...,11, find a series in which each pitch-class occurs exactly once and in which the musical intervals between neighboring notes cover the full set of intervals from the minor second (1 semitone) to the major seventh (11 semitones). That is, for each of the intervals, there is a pair of neighboring pitch-classes in the series, between which this interval appears. The problem of finding such a series can be easily formulated as an instance of a more general arithmetic problem on $\mathbb{Z}/n\mathbb{Z}$, the set of integer residues modulo $n$. Given $n$ in $\mathbb{N}$, find a vector $\vec{s} = \langle s_1,...,s_n \rangle$ such that (i) $\vec{s}$ is a permutation of $\mathbb{Z}/n\mathbb{Z} = \{0,1,...,n-1\}$; and (ii) the interval vector $\vec{v} = \langle |s_2-s_1|, |s_3-s_2|,..., |s_n-s_{n-1}| \rangle$ is a permutation of $\mathbb{Z}/n\mathbb{Z} \setminus \{0\} = \{1,2,...,n-1\}$. A vector $\vec{v}$ satisfying these conditions is called an all-interval series of size $n$; the problem of finding such a series is the all-interval series problem of size $n$. We may also be interested in finding all possible series of a given size.

The $\mathcal{F}$ model of the All-interval Series problem in Figure 7.14 captures the two permutations in the problem description by two bijections: $F$ is from $\vec{s}$ (representing $\vec{Z}_n$) into $\mathbb{Z}/n\mathbb{Z}$ and $G$ is from $\vec{v}$ (representing $\vec{u}$) into $\mathbb{Z}/n\mathbb{Z} \setminus \{0\}$. We also enforce that each $G(i)$ is equal to the absolute value of $F(i+1) - F(i)$.

Langford's number problem

Consider two sets of the numbers from 1 to 4. The problem is to arrange the eight numbers in the two sets into a single sequence in which the two 1's appear one number apart, the two 2's appear two numbers apart, the two 3's appear three numbers apart, and the two 4's appear four numbers apart.

The problem generalizes to the $L(n,m)$ problem, which is to arrange $n$ sets of numbers 1 to $m$, so that each appearance of the number $i$ is $i$ numbers on from the last. For example, the $L(3,9)$ problem is to arrange 3 sets of the numbers 1 to 9 so that the first two 1's and the second two 1's appear one number apart, the first two 2's and the second two 2's appear two numbers apart, etc.

An $\mathcal{F}$ model of the $L(n,m)$ problem is shown in Figure 7.15. The inputs are captured by the integers $n$ and $m$, the set of digits $\text{digits}$, the set of repetition $\text{rep}$, the set of occurrences of the digits $\text{occ}$, and the set of positions $\text{pos}$. The desired output is captured by a bijection $\text{Arrange}$ from $\text{occ}$ into $\text{pos}$. Finally, for each digit $i$ we have ordering constraints on its occurrences as well as restricting the difference between those occurrences to be $i$. However, we observe that the $\mathcal{F}$ model is not straight-forward to write.
n:int; m:int;
digits:{int} % digits={1,2,..,m}
rep:{int} % rep={1,2,...,n}
occ: {int}; % occ={1,2,...,n*m}
pos: {int} % pos={1,2,...,n*m}

var Arrange:occ->pos;
solve{
  bijective(Arrange);
  forall(i in digits)
    forall(j in rep: j <n)
      { Arrange(i+(j-1)*m) < Arrange(i+j*m);
        Arrange(i+j*m)- Arrange(i+(j-1)*m) = i;
      };
};

Figure 7.15: An \( F \) model of the \( L(n,m) \) problem

Bioinformatics

We consider the Word Design for DNA Computing on Surfaces problem (Prob033) as our example of a bioinformatics problem.

Word design for DNA computing on surfaces. This problem has its roots in bioinformatics and coding theory. The problem is to find as large a set \( S \) of strings (words) of length 8 over the alphabet \( W = \{A, C, G, T\} \) with the following properties:

- Each word in \( S \) has 4 symbols from \( \{A, C, G, T\} \);
- Each pair of distinct words in \( S \) differ in at least 4 positions; and
- Each pair of words \( x \) and \( y \) in \( S \) (where \( x \) and \( y \) may be identical) are such that \( x^R \) and \( y^C \) differ in at least 4 positions. Here, \( (x_1,\ldots,x_8)^R = (x_8,\ldots,x_1) \) is the reverse of \( (x_1,\ldots,x_8) \) and \( (y_1,\ldots,y_8)^C \) is the Watson-Crick complement of \( (y_1,\ldots,y_8) \), i.e. the word where each \( A \) is replaced by a \( T \) and vice versa and each \( C \) is replaced by a \( G \) and vice versa.

The natural way to view the desired output, based on the problem description, would be a set, and even when the desired output is instead viewed as a partial function from the set of all possible words into the set \( \{0,1\} \), it is still very difficult to express some of the problem constraints.

Games and puzzles

We consider the magic squares problem (Prob019) as our example of a puzzle.

Magic squares. An order \( n \) Magic Square is a \( n \) by \( n \) matrix containing the numbers 1 to \( n^2 \), with each row, column and main diagonal equal the same sum. As well as finding magic squares, we are interested in the number of a given size that exist.

Similar to the BIBD and the bioinformatics problem, a function variable is not a natural way of capturing the desired output of the Magic Squares problem. We might
think of a function variable from the set of entries in the square into the set of integers from 1 to \( n^2 \), but stating the problem constraints would not be easy. A more natural way is to define the desired output as a 2d array of integer variables, which allows an straightforward statement of the problem constraints as opposed to a function model.

**Summary on the scope of \( F \)**

The level of abstraction provided by \( F \) allowed us to easily express 7 out of the 10 problems, discussed above, as function models. The primitives of \( F \) were enough to cover a wide range of applications from different domains. The operations and constraints on function variables used in the above function models are:

- The function application operation;
- The inverse function operation appears in three contexts:
  1. a bound in a sum expression
  2. a parameter to the cardinality operator
  3. a parameter to the function application operator
- The membership predicate in functions was used to iterate over all elements in a function;
- The range of a function appeared in two contexts:
  1. a parameter to the cardinality operator
  2. a parameter of the max operator
- The bijective constraint;
- The injective constraint.

As for the BIBD, the bioinformatics, and the Magic Squares problems, the \( F \) language should be extended to allow variables of matrix, set and relation types along with their allowed operations in order to be able to model them. We should also extend \( F \) to allow sets of tuples to be either the source or target sets in a function variable. Extending the \( F \) languages with these extra data-types and other types will be further discussed in our future work section.

### 7.3.2 Quality of the generated \( L \) models

In this section, we focus on evaluating the quality of the generated \( L \) models. First, we carefully analyze the modelling decisions that the Fiona system makes while rewriting an \( F \) model into alternate \( L \) models. Second, we empirically demonstrate that there exists no best \( L \) model, i.e., for some function problems the \( L \) models that are based on the 1d array representation perform the best, while for others either the models based on the 2d 0/1 array representation or the combined models perform the best. Finally, we compare generated \( L \) models of some problems to already published hand-crafted ones.
Analysis of the modelling decisions

Throughout this thesis, our concern was mainly on the choice of $L$ variables for the $F$ variables, as well as the mapping of the $F$ constraints and expressions into $L$ constraints and expressions. It is worth noting that rewriting the $F$ expressions and constraints is dependent on our choice of representation for the $F$ variables.

We first start by the choice of representation for a function variable. Assuming we are seeking a total function from set $V$ into set $W$, our choice of representation can be classified into 4 categories:

- A 1d array of integer variables, indexed by $V$ and ranging over $W$: This representation is compact and commonly used by many modelers (see [34, 23, 43, 48] for examples). Many global constraints (e.g., alldifferent, atleast, max, etc) work on such kind of representation. However, one major drawback of this representation is that the set of elements of $V$ that have the same image is implicit in the representation. Except for special cases where a global constraint can be employed to express certain constraints (e.g., capacity constraints), more variables need to be introduced in order to be able to express the constraints on such sets (e.g., weighted-capacity constraints).

- A 2d array of Boolean variables indexed by $V$ and $W$ and row-sum constraints: This representation is commonly used in operations research. In this representation more variables are used than the previous one, but the variables have smaller domains. In most of the cases, linear constraints can be stated on this representation. However, its major drawback is that the constraints may be of high arity because to encode that an element $i$ in $V$ has an image $j$ in $W$, we employ $|W|$ Boolean variables. When all the constraints on this representation are linear, we can use an ILP solver to solve the resulting model.

- A 1d array, indexed by $W$, of set variables that are subsets of $V$ such that their union is $V$ and they are pairwisely disjoint: This representation allows the exploitation of global constraints developed for set variables. However, the image of each element in $V$ is implicit in this representation, which may result in introducing more variables in order to state some constraints that involve the function application operation.

- Combined representations:
  - We combine the first and the second representation in order to cure the drawbacks of each representations. The second one explicitly represents the set of elements of $V$ that have the same image, while the first representation allows the statement of constraints of fewer arity.
  - This combined representation uses the first and the third representation. By doing so, we are trying to overcome the problem of the first representation by using the third representation, and vice versa. In the combined representation all aspects of the function are explicitly represented, which allows the ease of the statement of the problem constraints.
  - One can also overcome the problems of the third representation by using the third and the second one. As a result we may also use the third representation to state global constraints instead of higher arity constraints on the second representation.
Our last combined representation employs both primal and dual variables at the same time. This is exploited in the cases of bijection and injection problems. For bijection problems, we exploit the results established by Cheng et al. [12, 13], Smith [66, 67], and Walsh [81]. We also produce similar results for injection problems.

The major drawback of the combined representations is the increase in the number of variables and the addition of extra channelling constraints.

When the function is partial instead, we only consider modifications of the three basic representations for a total function by introducing extra variables that represent the domain of the partial function. In addition to the increase of number of variables, the constraints on the first representation may be of high arity. Another alternative for partial functions would be to transform them to a total function. However, we fail to do this automatically because the elements of the target set may have certain semantics and adding a dummy value cannot be random then. Nevertheless, the user can model his partial function as a total one and thus all the representations developed for a total function would be exploited.

As for the constraint formulation and constraint formulation selection heuristics for integrated/hybrid models, we have carefully exploited, throughout Chapter 5 and Chapter 6, every opportunity to best state the problem constraints:

- We employ a global constraint whenever it is possible. For instance, the $L$ model in Figure 7.4 uses three global constraints, namely, \texttt{atleast}, \texttt{atmost}, and \texttt{max}. Similarly, the $L$ model in Figure 7.6 employs the cardinality global constraint (\texttt{card}), the weighted-cardinality global constraint (\texttt{weighted-cardinality}), and the global constraint \texttt{max}.

- We try to state linear constraints whenever the model is to be solved with an ILP solver. For instance, all the constraint in the $L$ model in Figure 7.5 are linear.

- We prefer to state fewer constraints as demonstrated in the case of linking constraints in Chapter 6, where we decided to link the extra variables representing the range of the injection to the dual variables rather than the primal ones because we it uses fewer constraints.

- We choose constraints of lesser arity whenever it is possible. For instance, in the $L$ models in Figure 7.7 and in Figure 7.9, we state the prerequisite constraints using the formulation on $F_{d1}$ because the constraints are of lesser arity than those stated on $F_{d2}$ and on $F_S$.

- We developed the constraint formulation selection heuristics so as to pick the better constraint formulation, whenever possible.

Therefore, in the resulting $L$ models, the best constraint formulation— to the best of our knowledge— has been employed. Indeed, the generated $L$ models of the BACDP are the same as to hand-crafted ones developed in [11] and [43]. Also, the hand-crafted models of the (optimization) GCP in Chapter 4 are generated by the Fiona system.
maintcost : int;
stores: {int};
warehouses : {int};
capacity : warehouses -> int;
supplycost: stores x warehouses -> int;
var int SUPPLIER_d1[stores] in warehouses;
var int range_SUPPLIER[warehouses] in 0..1;
minimize
sum(i in stores) supplycost(<i,SUPPLIER_d1[i]>)+
sum(j in warehouses)range_SUPPLIER[j]*maintcost
subject to{
%linking constraints
forall(i in stores) forall(j in warehouses)
  SUPPLIER_d1[i]=j => range_SUPPLIER[j]=1;
forall (j in warehouses) atmost(j,SUPPLIER_d1,capacity(j));
}

Figure 7.16: A generated \( \mathcal{L} \) model of the WLP using 1d array

No best \( \mathcal{L} \) model

We recall that for the BACDP, the fastest model to prove optimality is the combined one solved with a hybrid solver. We also showed in Chapter 6 that for injection problems, the performance of the models varied from one problem to the other. For some, the models using the \textit{alldifferent} constraint were the fastest, while on other problems, the model using the channelling constraints (and the implied constraints) were as good as the ones that use the \textit{alldifferent} constraint in terms of failures, and on loose problems, models that use the binary not-equals constraints were the fastest. Now we carry out more experiments on a number of other problems.

Warehouse location problem. Among the possible \( \mathcal{L} \) models of the WLP, we show a model based on the 1d array representation in Figure 7.16, a model based on a 2d 0/1 array in Figure 7.17, and a hybrid one combining both in Figure 7.18. We implemented them, in a straightforward manner, in \textit{opl}. We used a CP solver for the first model, an ILP solver for the second, and a hybrid one for the third.

We tried the three different models on the real-life instance data (cap44 from the operations research library (ORLIB)), where we used a lexicographical variable ordering heuristic and a numerical value ordering heuristic. The fastest model was the ILP model shown in Figure 7.17 (12.49 seconds), followed by the hybrid model shown in Figure 7.18 (31.3 seconds), and last the CP model shown in Figure 7.16 (>100 seconds). Also, the hybrid model achieved more pruning than the pure CP one. We tried larger instances, but with our variable and value ordering heuristics, all models couldn’t solve these instances within 1 hour.

Bus driver scheduling problem. The generated \( \mathcal{L} \) models that do not use set variables are shown in Figure 7.19, Figure 7.20, and Figure 7.21. In a straightforward manner, we implemented them in \textit{opl}. We used a CP solver to solve the one in Figure 7.19, an
maintcost : int;
stores: {int};
warehouses : {int};
capacity :warehouses -> int;
supplycost:stores x warehouses -> int;
var int SUPPLIER_d1[stores] in warehouses;
var int SUPPLIER_d2[stores,warehouses] in 0..1;
var int range_SUPPLIER[warehouses] in 0..1;
minimize
  sum(i in stores, j in warehouses) SUPPLIER_d2[i,j]*supplycost(<i,j>)+
  sum(j in warehouses)range_SUPPLIER[j]*maintcost
subject to{
  %every i in stores gets assigned exactly 1 element in warehouses
  forall(i in stores) sum(j in warehouses) SUPPLIER_d2[i,j]=1;
  %linking constraints
  forall(i in stores) forall(j in warehouses)
  SUPPLIER_d2[i,j] <= range_SUPPLIER[j];
  forall (j in warehouses) sum(i in stores) SUPPLIER_d2[i,j]<=capacity(j)};

Figure 7.17: A generated $\mathcal{L}$ model of the WLP using 2d 0/1 array

ILP solver for the model in Figure 7.20, and a hybrid one for the one in Figure 7.21. But, note that we had to manually rewrite the constraint in Figure 7.20 because it is not linear:

\[
\text{range}_\text{Schedule}[j]=1 \Rightarrow \sum(i \text{ in pieces}) \text{ Schedule}_d2[i,j] = \text{card(coverage(j))}
\]

into this linear constraint instead:

\[
\sum(i \text{ in pieces}) \text{ Schedule}_d2[i,j] = \text{card(coverage(j))}*\text{range}_\text{Schedule}[j]
\]

We plan to automate such kind of transformations in our future work.

We tried the three different models on the real-life instance data (t1) in [16], using a lexicographical variable ordering heuristic and numerical value ordering heuristic. The fastest model was the CP model shown in Figure 7.19 (1.79 seconds and 294 failures), followed by the ILP model shown in Figure 7.20 (4.68 seconds), and last the hybrid model shown in Figure 7.21 (54.32 seconds and 10 failures). However, in terms of failures, the hybrid model achieved less number of failures than the CP model. We also tried bigger instances, but none of the models solved them in a reasonable amount of time (1 hour).

**Langford’s problem.** We consider comparing the following $\mathcal{L}$ models generated from the $\mathcal{F}$ model in Figure 7.15:

- The model Fd1+MGACV is shown in Figure 7.22 and uses a 1d array plus an alldifferent constraint where GAC is maintained during search.
maintcost : int;
stores: {int};
warehouses : {int};
capacity : warehouses -> int;
supplycost:stores x warehouses -> int;
var int SUPPLIER_d2[stores,warehouses] in 0..1;
var int range_SUPPLIER[warehouses] in 0..1;
minimize
  sum(i in stores, j in warehouses) SUPPLIER_d2[i,j]*supplycost(<i,j>)+
  sum(j in warehouses)range_SUPPLIER[j]*maintcost
subject to{
%every i in stores gets assigned exactly 1 element in warehouses
  forall(i in stores) sum(j in warehouses) SUPPLIER_d2[i,j]=1;
%Channelling constraints
  forall(i in stores, j in warehouses)
    SUPPLIER_d1[i]=j <= SUPPLIER_d2[i,j]=1;
%linking constraints
  forall(i in stores) forall(j in warehouses)
    SUPPLIER_d2[i,j] <= range_SUPPLIER[j];
%linking constraints
  forall(i in stores) forall(j in warehouses)
    SUPPLIER_d1[i]=j => range_SUPPLIER[j]=1;
  forall (j in warehouses) atmost(j,SUPPLIER_d1,capacity(j));
/* Use global constraint when a CP solver is used */
/* When a CP+ILP solver is used also use this linear formulation: */
/* sum(i in stores) SUPPLIER_d2[i,j] <=capacity(j)*/ };

Figure 7.18: A hybrid L model of the WLP

* The model Fd1+MACc is shown in Figure 7.23 and introduces a dual array and uses channelling constraints. AC is maintained during search on the channelling constraints.

* The model Fd1+MACf is shown in Figure 7.24 and uses a 1d array and binary not-equals constraints. AC is maintained during search on the not-equals constraints.

* The model Fd2 is shown in Figure 7.25 and uses a 2d 0/1 array.

* The model Fd1+Fd2 is shown in Figure 7.26 and is the combined model of the one in Figure 7.22 and the one in Figure 7.25.

We implemented the different models in sicstus prolog, and the results are shown in Table 7.3. The worst model in terms of runtime and failures is model Fd2. The fastest model is Fd1+MACc, followed by Fd1+MGACV, followed by Fd1+MACf, and followed by Fd1+Fd2. However, in terms of failures, Fd1+MGACV, Fd1+MACc, Fd1+Fd2 recorded the same value followed by Fd1+MACf.
**Figure 7.19**: Generated $\mathcal{L}$ model of the bus driver scheduling problem using 1d array

**Table 7.3**: No. of backtracks (fails) and running time (in milliseconds) to find all solutions to four instances of Langford’s problem. A dash means that no results were collected after 1 hour.

**Comparison with hand-crafted models**

In [65], three models of Langford’s problem are presented. The “primal all-different” model, which is the same as our model presented in Figure 7.22, The “primal not-equals” model, which is the same as our model in Figure 7.24, and finally a “minimal combined” model, which is the same as our model shown in Figure 7.23.

Starting from the $\mathcal{F}$ model in Figure 7.13, our system generates four $\mathcal{L}$ models shown in Figure 7.27, in Figure 7.28, in Figure 7.29, and in Figure 7.30. Our model in Figure 7.27 is the same as the “ternary and all-different” model discussed in [69]. Similarly, our model in Figure 7.29 is the same as the “ternary and not-equals” model presented in [69].

In [75], three models of the WLP are presented. Ignoring the labelling heuristics, our generated model in Figure 7.16 is the same as the one on page 220 of [75]. The only difference between our model in Figure 7.17 and the one on page 224 of [75] is our statement of the capacity constraint by means of a global constraint whereas the one in page 224 of [75] uses reification constraints. Finally, our generated model in Figure 7.18 differs from the one on page 225 of [75] in the statement of the capacity constraints on the 1d array and on
pieces: {int};
shifts: {int};
coverage: shifts->{int};
var int Schedule_d2[pieces,shifts] in 0..1;
var int range_Schedule[shifts] in 0..1;
minimize
  sum(i in shifts) range_Schedule[i]
subject to {
  %every i in pieces gets assigned exactly 1 element in shifts
  forall(i in pieces) sum(j in shifts) Schedule_d2[i,j]=1;
  %linking constraints
  forall(i in pieces) forall(j in shifts)
    Schedule_d2[i,j] <= range_Schedule[j];
  forall(j in shifts)
    forall(i in pieces: i not in coverage(j))
      Schedule_d2[i,j]=0;
  forall(j in shifts)
    range_Schedule[j]=1 =>
      sum(i in pieces) Schedule_d2[i,j] = card(coverage(j));
}

Figure 7.20: Generated \( L \) model of the bus driver scheduling problem using a 2d 0/1 array

the 2d 0/1 array. We use a global constraint on our 1d array while reification constraints are used for the one on page 225 of [75]. We state the capacity constraints in our model on the 2d 0/1 array as \( \sum(i \text{ in stores}) \text{SUPPLIER}_d2[i,j] \leq \text{capacity}(j) \) while they are stated as \( \sum(i \text{ in stores}) \text{SUPPLIER}_d2[i,j] \leq \text{capacity}(j) \times \text{rangeSUPPLIER}[j] \) on the model on page 225 of [75].

Summary on Fiona

We have argued that the Fiona system allows different representation choices for the \( F \) variables and for a given choice, we exploit every opportunity to best state the problem constraints. Also, we empirically demonstrated that there exists no best \( L \) model. For some function problems, the \( L \) models that are based on the 1d array representation perform the best, while for others either the models based on the 2d 0/1 array representation or the combined models perform the best. Furthermore, we notice a high level of similarity between the generated \( L \) models of some problems and the already published hand-crafted ones. However, for some problems, such as the bus driver scheduling problem, our generated models were only able to solve one instance of the problem. This is due to the fact that those models should be further enhanced by designing a good labelling strategy, adding symmetry-breaking and implied constraints, etc. Automating parts of these tasks remain an open research question.

Finally, it is worth pointing out that the Fiona system is a very flexible system that can easily be changed and extended as we gain more insights on how to better choose our variables and on how to efficiently formulate our constraints:

136
pieces: {int};
shifts: {int};
coverage: shifts->{int};
var int Schedule_d1[pieces]in shifts;
var int Schedule_d2[pieces,shifts] in 0..1;
var int range_Schedule[shifts] in 0..1;
minimize
  sum(i in shifts) range_Schedule[i]
subject to {
  %every i in pieces gets assigned exactly 1 element in shifts
  forall(i in pieces) sum(j in shifts) Schedule_d2[i,j]=1;
  %Channelling constraints
  forall(i in pieces, j in shifts)
    Schedule_d1[i]=j => Schedule_d2[i,j]=1;
  %linking constraints
  forall(i in pieces) forall(j in shifts)
    Schedule_d1[i]=j => range_Schedule[j]=1;
  %linking constraints
  forall(i in pieces) forall(j in shifts)
    Schedule_d2[i,j] <= range_Schedule[j];
  forall(j in shifts)
    forall(i in pieces: i not in coverage(j))
      Schedule_d1[i] <> j;
  forall(j in shifts)
    range_Schedule[j]=1 =>
      sum(i in pieces) Schedule_d2[i,j] = card(coverage(j));
}

Figure 7.21: Generated L model of the bus driver scheduling problem using a 1d array and a 2d 0/1 array

- More rewrite rules can be added to capture more special cases where a more “efficient” representation or a more “efficient” constraint formulation can be exploited.
- The current rewrite rules may be changed or improved.
- More constraint formulation selection heuristics can be developed and incorporated in the rewrite rules.
- The conditions in the rewrite rules may be strengthened by considering the information provided by the instance data, for instance.

7.4 Extensions

The tool may be extended in many different ways:

- Other tools presented in the literature, such as Borrett and Tsang’s methods for the automatic generation and evaluation of ICs [5], or Frisch et al.’s tool that generates ICs and symmetry-breaking constraints [32], can easily be added as new sub-components within the proposed tool. Such tools will take as input the models
var int Arrange_d1[occ] in pos;
solve{
    alldifferent(Arrange_d1);
    forall(i in digits)
        forall(j in rep: j < n)
            { Arrange_d1[i+(j-1)*m] < Arrange_d1[i+j*m];
              Arrange_d1[i+j*m]- Arrange_d1[i+(j-1)*m] = i;
            };
};

Figure 7.22: A Generated L model of the L(n,m) problem

generated by our tool and generate more models that employ more ICs or symmetry-breaking constraints.

• The work done by Minton [54] on tailoring labelling strategies to particular instance distributions can be added as a further extension to the tool.

• One can always choose a target CP language and generate only the models that can be expressed in that language [26, 28].

Other extensions will be discussed in Chapter 8

7.5 Summary

In this chapter, an architecture of a practical modelling tool is presented. The tool serves the goal of helping modelers explore different models of function problems that are expressed at a high-level of abstraction. The tool will help inexperienced users gain some knowledge about modelling issues, as well as speed up the process of modelling for an expert user. We believe that such tools are necessary in the area of constraint programming. Such a tool is a result of identifying many recurring modelling idioms in constraint programs, generalizing them, and providing high level constructs to capture them. These modelling idioms are then reproduced automatically from these high level constructs. Thus, the tool constitutes a step towards revealing the secrets behind the design of good models for a function problem.

138
n:int;
m:int;
digits:{int} % digits={1,2,...,m}
rep:{int} % rep={1,2,...,n}
occ: {int}; % occ={1,2,...,n*m}
pos: {int} % pos={1,2,...,n*m}
var int Arrange_d1[occ] in pos;
var int DArrange_d1[pos] in occ;
solve{
    forall(i in occ, j in pos)
        Arrange_d1[i]=j <=> DArrange_d1[j]=i;
    forall(i in digits)
        forall(j in rep: j <n)
            { Arrange_d1[i+(j-1)*m] < Arrange_d1[i+j*m];
                Arrange_d1[i+j*m]- Arrange_d1[i+(j-1)*m] = i;
            };
};

Figure 7.23: A Generated $\mathcal{L}$ model of the L($n,m$) problem

n:int;
m:int;
digits:{int} % digits={1,2,...,m}
rep:{int} % rep={1,2,...,n}
occ: {int}; % occ={1,2,...,n*m}
pos: {int} % pos={1,2,...,n*m}
var int Arrange_d1[occ] in pos;
solve{
    forall(i in digits)
        forall(j in rep: j <n)
            { Arrange_d1[i+(j-1)*m] < Arrange_d1[i+j*m];
                Arrange_d1[i+j*m]- Arrange_d1[i+(j-1)*m] = i;
            };
};

Figure 7.24: A Generated $\mathcal{L}$ model of the L($n,m$) problem
n:int;
m:int;
digits:{int} % digits={1,2,...,m}
rep:{int} % rep={1,2,...,n}
occ: {int}; % occ={1,2,...,n*m}
pos: {int} % pos={1,2,...,n*m}
var int Arrange_d1[occ] in pos;
var int Arrange_d2[occ,pos] in 0..1;
solve{
%every i in pieces gets assigned exactly 1 element in shifts
forall(i in occ) sum(j in pos) Arrange_d2[i,j]=1;
%Channelling constraints
forall(i in occ, j in pos) Arrange_d1[i]=j <=> Arrange_d2[i,j]=1;
alldifferent(Arrange_d1);
forall(i in digits)
forall(j in rep: j <n)
{sum(k in pos) Arrange_d2[i+(j-1)*m,k]*k <
 sum(k in pos) Arrange_d2[i+j*m,k]*k;
 sum(k in pos) Arrange_d2[i+j*m,k]*k -
 sum(k in pos) Arrange_d2[i+(j-1)*m,k]*k = i;} };

Figure 7.25: A Generated $L$ model of the $L(n,m)$ problem

n:int;
m:int;
digits:{int} % digits={1,2,...,m}
rep:{int} % rep={1,2,...,n}
occ: {int}; % occ={1,2,...,n*m}
pos: {int} % pos={1,2,...,n*m}
var int Arrange_d1[occ] in pos;
var int Arrange_d2[occ,pos] in 0..1;
solve{
%every i in pieces gets assigned exactly 1 element in shifts
forall(i in occ) sum(j in pos) Arrange_d2[i,j]=1;
%Channelling constraints
forall(i in occ, j in pos) Arrange_d1[i]=j <=> Arrange_d2[i,j]=1;
alldifferent(Arrange_d1);
/* Use global constraint when a CP solver is used */
/* When a CP+ILP solver is used also use this linear formulation: */
forall(j in pos) sum(i in occ) Arrange_d2[i,j]=1*/
forall(i in digits)
forall(j in rep: j <n)
{ Arrange_d1[i+(j-1)*m] < Arrange_d1[i+j*m];
 Arrange_d1[i+j*m]- Arrange_d1[i+(j-1)*m] = i; }
};

Figure 7.26: A Generated $L$ model of the $L(n,m)$ problem
ticks: {int}; % ticks={1,...,m}
distances: {int}; % distance has m(m-1)/2 values
pair:distance->ticks x ticks;
% every distance has an image an ordered pair of ticks
var int Tick-assign_d1[ticks] in 0..maxint;
var int Distance-assign_d1[distances] in 0..maxint;
minimize
  maxrange
subject to {
  Tick-assign_d1[1]=0;
  forall(i in ticks: i<m)
    Tick-assign_d1[i] < Tick-assign_d1[i+1];
  alldifferent(Tick-assign_d1);
  alldifferent(Distance-assign_d1);
  forall(<a,<b,c>> in pair)
    Distance-assign_d1[a] = Tick-assign_d1[b] - Tick-assign_d1[c];
  max(Tick-assign_d1,maxrange)};

Figure 7.27: A generated $L$ model of the golomb ruler problem

ticks: {int}; % ticks={1,...,m}
distances: {int}; % distance has m(m-1)/2 values
pair:distance->ticks x ticks;
% every distance has an image an ordered pair of ticks
var int Tick-assign_d1[ticks] in 0..maxint;
var int Distance-assign_d1[distances] in 0..maxint;
minimize
  maxrange
subject to {
  Tick-assign_d1[1]=0;
  forall(i in ticks: i<m)
    Tick-assign_d1[i] < Tick-assign_d1[i+1];
  forall(ordered i,j in ticks) Tick-assign_d1[i] <> Tick-assign_d1[j];
  alldifferent(Distance-assign_d1);
  forall(<a,<b,c>> in pair)
    Distance-assign_d1[a] = Tick-assign_d1[b] - Tick-assign_d1[c];
  max(Tick-assign_d1,maxrange)};

Figure 7.28: A generated $L$ model of the golomb ruler problem
ticks: {int}; % ticks={1,..,m}
distances: {int}; % distance has m(m-1)/2 values
pair:distance->ticks x ticks;
% every distance has an image an ordered pair of ticks
var int Tick-assign_d1[ticks] in 0..maxint;
var int Distance-assign_d1[distances] in 0..maxint;
minimize
    maxrange
subject to {
    Tick-assign_d1[1]=0;
    forall(i in ticks: i<m)
        Tick-assign_d1[i] < Tick-assign_d1[i+1];
    alldifferent(Tick-assign_d1);
    forall(ordered i,j in distances)
        Distance-assign_d1[i] <> Distance-assign_d1[j];
    forall(<a,<b,c>> in pair)
        Distance-assign_d1[a] = Tick-assign_d1[b] - Tick-assign_d1[c];
    max(Tick-assign_d1,maxrange)};

Figure 7.29: A generated $\mathcal{L}$ model of the golomb ruler problem

ticks: {int}; % ticks={1,..,m}
distances: {int}; % distance has m(m-1)/2 values
pair:distance->ticks x ticks;
% every distance has an image an ordered pair of ticks
var int Tick-assign_d1[ticks] in 0..maxint;
var int Distance-assign_d1[distances] in 0..maxint;
minimize
    maxrange
subject to {
    Tick-assign_d1[1]=0;
    forall(i in ticks: i<m)
        Tick-assign_d1[i] < Tick-assign_d1[i+1];
    alldifferent(Tick-assign_d1);
    forall(ordered i,j in ticks)
        Tick-assign_d1[i] <> Tick-assign_d1[j];
    forall(ordered i,j in distances)
        Distance-assign_d1[i] <> Distance-assign_d1[j];
    forall(<a,<b,c>> in pair)
        Distance-assign_d1[a] = Tick-assign_d1[b] - Tick-assign_d1[c];
    max(Tick-assign_d1,maxrange)};

Figure 7.30: A generated $\mathcal{L}$ model of the golomb ruler problem
Chapter 8

Conclusion

We first summarize our contributions in Section 8.1. Then we discuss the results of the thesis in Section 8.2. Finally, in Section 8.3, we present our plans for future work.

8.1 Summary

Modelling a problem as a CSP in an efficient and effective manner requires a lot of skills. Furthermore, in practice, more than one model has to be tried in order to find a good problem representation. Except for a few methods that try to automate some of the tasks in modelling, such as [5, 4, 32] for the automatic generation and evaluation of ICs, very little work has been done in the direction of helping problem modeler’s with their task.

This thesis is based on the belief that there is a great need for tools that would assist the modeler in exploring alternate models, as well as in deciding which of the models to select. Towards achieving this goal, the modelling process need to be dissected and its parts need to be independently highlighted and studied.

In a CSP, the goal is to assign values to variables from their domains such that some constraints are satisfied. Each constraint is a relation defining the allowed values for a given subset of variables. A solution to a CSP is an assignment of values to the variables that is consistent with all constraints. It follows from the definition of a CSP that two major decisions need to be made:

- The choice of variables, their domains, and the constraints.
- The choice of a solution method, which consists of choosing a search algorithm and propagation algorithms that enforce certain consistency level on the constraints.

The first item is referred to as a model while the second item is referred to as a solver. By applying a solver to a model a solution to the problem can be computed.

Constraint programming (CP) is a two-level architecture. The first level provides a language to specify the variables, their domains, and the constraints. The second level is composed of a solver. In general, the solver is parameterized by the labelling strategy, i.e., which variable to branch on next and which value to assign to that variable. Different CP languages have different characteristics. Some differ at the first level while others differ at the second level. For instance, at the first level, some CP languages allow the declaration of only integer variables, others allow also the declaration of set variables. At the second level, some use the forward-checking search algorithm, while others maintain generalized arc-consistency during search.
In order to be able to provide assistance to modelers, these two important questions need to be addressed:

1. How easy it is to develop a model for a given problem?

2. How efficiently can we solve the resulting model?

To answer the first question, we need just to analyze what the language at the first level offers in terms of variable types, their domains, and constraints. The richer the language is, the easier the task becomes. To answer the second question, we need to understand the interaction between the solver and the model. This requires, in part, knowing which constraint formulation is efficiently handled in the solver and which is not. Trying to answer this question may force us to rethink about our models. Sometimes we have to restate the same constraint differently, while in some other cases we add implied constraints to the model or change our variable modelling, etc.

In this thesis, two languages \( L \) and \( F \) have been introduced. The language \( L \) is an attempt to provide a concise notation that allows the description of constraint programs independently of any current CP language. The language \( L \) allows reasoning about problems in terms of (array of) integer and set variables. The domains can either be integer sets or sets of integer sets. The language \( L \) restricts the set of constraints that can possibly be formulated on variables of integer and set types in such a way that it reflects what is possible in some current CP languages. A set of global constraints are also supported in \( L \). The purpose of the \( L \) language is to allow the description of the modelling decision made at the first level of current CP languages. Thus, an \( L \) model of a given problem represents the desired output in terms of (arrays of) integer and set variables and encodes the problem constraints as constraints on these variables using arithmetic and set operations. Introducing \( L \) gave us a degree of freedom of what a model can be composed of without introducing extra features that are not supported in any current CP language. For example, in an \( L \) model, set variables and integer variables can be declared at the same time, which is only possible in languages like \texttt{oz}, \texttt{eclipse}, and \texttt{ilog solver}, but not in languages like \texttt{opl} and \texttt{sicstus prolog}.

The language \( F \) on the other hand is very different from \( L \). The language \( F \) allows reasoning about problems in terms of functions, permutations, sequences, and their operations. Decision variables can only be of function, permutation, or sequence type. Constraints on variables of these types are stated with the help of function, permutation, and sequence operations in addition to arithmetic and set operations. There are two major differences between \( F \) and \( L \). First, \( F \) allows decision variables to have as domain the set of all possible functions from a given set into another given set, all possible permutations over a given set, or all possible sequences (of fixed or bounded length) over a given set. Second, function, permutation, and sequence operations may also be used to state the constraints of the problem. Thus, an \( F \) model of a given problem, as opposed to an \( L \) model, represents the desired output in terms of function, permutation, and sequence variables and encodes the problem constraints as constraints on these variables using arithmetic, set, function, permutation, and sequence operations.

In Chapter 4, we introduced function, permutation, and sequence variables along with the allowed operations on variables of these types. We have shown that there is a number of problems that can be concisely expressed in \( F \). These problems are called function problems because the objective is to find a function from a given set into another given set. We have shown through the Graph Coloring problem that an \( F \) model of the problem, where the only decision variable is a function variable from the set of vertices into the
set of colors, is easier to update than two \( \mathcal{L} \) models that represent the desired output in terms (arrays of) of integer variables. The Warehouse Location problem shows that maintaining an \( \mathcal{F} \) model, when there is a minor change in the requirements, is straightforward, while we had to introduce more variables and more constraints in one of the \( \mathcal{L} \) models and reformulate some of the already stated constraints in the other one. In the last problem of Chapter 4, the Balanced Academic Curriculum Design problem, our attention was concerned with efficiently solving three real-life instances of the problem. We have shown that alternate \( \mathcal{L} \) models when considered with respect to a certain solver vary in efficiency. The best models were combined models that either use the same solver (CP) or a hybrid solver (CP and ILP). The proposed \( \mathcal{F} \) model on the other hand is a natural formulation of the problem that is much easier to understand than all proposed \( \mathcal{L} \) models.

In Chapter 5, we showed how in practice function, sequence, and permutation variables are represented in different ways in terms of arrays of integer and set variables. We considered three possible representations of a function from a given source set into a given target set; the first representation uses an array of integer variables indexed by the source set and ranging over the target set, the second uses a 2 dimensional array of Boolean variables indexed by the source and target sets plus row-sum constraints, while the third uses an array of set variables that are constrained to be pairwisely disjoint and their union to be the domain set of the function. We also showed how to mechanically map constraints on these variables into constraints on integer and set variables depending on the representation choice. We mapped some constraints expressed in \( \mathcal{F} \) into different constraints in \( \mathcal{L} \) even on the same representation. This is sometimes achieved by introducing dual variables and properly channelling them with the primal variables (e.g., when enforcing the bijectiveness and injectiveness constraints). We also demonstrated how a permutation and a sequence variable can be seen as a function variable with some extra constraints. We then explored the possibility of combining any two of the considered representations of a function variable that is total. For the combined models that use the second representation, we proposed using either a CP solution method or a hybrid CP and ILP solution method, while for all other combined models, a CP solution method was advocated. We analyzed different constraint formulation in a general way, and proposed the heuristics \( \mathcal{H}_{\text{arithmetic}} \), \( \mathcal{H}_{\text{inverse}} \), \( \mathcal{H}_{\text{range}} \), and \( \mathcal{H}_{\text{membership}} \) that choose which constraint formulation to use for the combined model. The heuristic \( \mathcal{H}_{\text{arithmetic}} \) recommends an \( \mathcal{L} \) constraint formulation for arithmetic constraints, expressed in an \( \mathcal{F} \) model, that involve function applications. The heuristic \( \mathcal{H}_{\text{inverse}} \) recommends an \( \mathcal{L} \) constraint formulation for constraints, expressed in an \( \mathcal{F} \) model, that involve inverse function applications. The heuristic \( \mathcal{H}_{\text{range}} \) recommends an \( \mathcal{L} \) constraint formulation for constraints, expressed in an \( \mathcal{F} \) model, that involve range set of a function variable. The heuristic \( \mathcal{H}_{\text{membership}} \) chooses an \( \mathcal{L} \) constraint formulation for constraints, expressed in an \( \mathcal{F} \) model, that involve the membership predicate allowed for functions. We also presented the channelling constraints that link the two sets of variables in a combined model. Finally, we exploited certain properties that certain functions must satisfy and by adding these properties as constraints in an \( \mathcal{F} \) model, a set of useful ICs is added to the generated \( \mathcal{L} \) models. The major contribution of Chapter 5 is that starting from an \( \mathcal{F} \) model, alternate \( \mathcal{L} \) models can be automatically generated. As a results, an \( \mathcal{L} \) modeler is released from making the following modelling decisions for a function problem:

- The choice of the variables and their domains;
• The formulation of the constraints;
• The introduction of appropriate dual variables;
• The introduction of channelling constraints;
• The integration of different representations;
• Choosing between different constraint formulations;
• The addition of some ICs.

The process of transforming models written in \( F \) generates, in general, more than one model in \( L \).

In Chapter 6 significant steps towards solving the question of automatic selection of models are proposed. We first exposed some of the elements that make the task of model selection difficult. Then, we proposed different models of injection problems and used the constraint tightness parameterized by the level of local consistency being enforced, proposed by Walsh [81], to compare these models. In the proposed primal models, the injectiveness constraints is enforced either by stating a global \( \text{alldifferent} \) constraint or binary not-equals constraints on the primal variables. Three ways of introducing dual variables are proposed and three types of channelling constraints are used to link the primal and dual variables. Further constraints can be stated on the dual variables when the second and third type of channelling constraints are used. When the second type of channelling constraints is used, binary not-equals constraints on the dual variables or a set of implied constraints can be further added. When the third type of channelling constraints is used, global occurs constraints on the dual variables or a set of implied constraints can be further added. We proved that, with respect to arc-consistency, forward checking, and bounds consistency, a single primal \( \text{alldifferent} \) constraint is tighter than the channelling constraints together with the implied, the occurs or the dual not-equals constraints, but that the channelling constraints alone are as tight as the primal not-equals constraints.

Both these gaps can lead to an exponential reduction in search cost when an algorithm that maintain (generalized) arc-consistency during search is used. The theoretical results showed that the global occurs constraints are redundant, so they can be safely discarded. We also analyzed the time required to achieve (generalized) arc-consistency on the different constraints of the models that survived the theoretical analysis. We concluded that it is safe to discard the model that uses the second type of channelling constraints as well as the model that adds the dual not-equals constraints to these channelling constraints because they achieve less pruning than the model that adds the implied constraints to the channelling constraints of the second type at the same cost. Similarly, we discarded the model that only uses the channelling constraints of the third type because it achieves less pruning than the model that adds the implied constraints to these channelling constraints at the same cost. However, we kept the model that uses the channelling constraints of the first type even though it achieves the same amount of pruning as the model that uses the primal not-equals constraints at a higher cost because there is an opportunity for achieving a cheap value ordering heuristic by developing a variable ordering heuristic on the dual variables. Experimental results on the modified 8-Queens problem and the Sport Scheduling problem confirmed that maintaining generalized arc-consistency during search on the channelling constraints of the second and third types together with the implied constraints could be competitive with maintaining generalized arc-consistency.
during search on the primal \textit{alldifferent} constraint. However, on less constrained problems such as the Graceful Graph problem maintaining arc-consistency during search on the primal not-equals constraints achieves comparable results with maintaining generalized arc-consistency during search on the primal \textit{alldifferent} constraint. Hence it might not be beneficial to have additional constraint propagation for such cases.

Finally, in Chapter 7, we put all our contributions together and proposed an architecture of a practical modelling tool. The tool serves the goal of helping modelers explore different models of function problems that are expressed in \( \mathcal{F} \). The tool will help inexperienced users gain some knowledge about modelling issues, as well as speeds up the process of modelling for an expert user. We believe that such tools are necessary in the area of constraint programming. Such a tool is a result of identifying, generalizing, and encoding many modelling tricks and reproducing them from abstract descriptions. Thus, the tool constitutes a step forward towards revealing the secrets behind the design of a good model. To the best of our knowledge, the proposed tool is the first attempt in the literature that addresses ways of simplifying the modelling of function problems as constraint programs.

8.2 Discussions

The thesis has the following limitations:

1. Reasoning about problems in terms of functions and function operations requires users to be familiar with such concepts. Even though, functions and their operations are well established concepts, it might take some time to be able to reason in terms of these concepts.

2. The set of generated \( \mathcal{L} \) models from an \( \mathcal{F} \) model of a function problem, where the desired function is partial do not contain models that add a dummy element to the target set, with the elements of the source set that are not in the domain of the function has the dummy value as an image. This technique transforms the problem from one where the objective is to find a partial function from a given set into another given set to a problem where the objective is to find a total function instead. This transformation technique may lead to better models as is the case for the Rack Configuration problem [48]. In this thesis, we fail to automate this transformation technique and hence miss some opportunity of generating models that might be better than the proposed ones.

3. The analysis of alternate \( \mathcal{L} \) models does not incorporate knowledge about the instance distribution, which might have a great impact on the efficiency of the resulting models as demonstrated in [73, 54]. This thesis does not make use of the instance distribution.

4. The thesis proposes a set of useful heuristics that choose between different constraint formulations, but the issue of conflict resolution, which arises when two or more heuristics apply to the same constraint but have conflicting recommendations, is addressed by generating all possible combinations rather than devising ways of ordering the impact of the recommendations of the different heuristics.

Except for the limitation presented in the first item, more work needs to be done in the future to address the remaining limitations.
8.3 Future work

There is a number of issues that will be addressed in our future work:

- **Extending the \( \mathcal{F} \) language:**
  - To make the \( \mathcal{F} \) language accessible to more users, a graphical notation may be developed. A Unified Modelling Language like graphical notation may be adopted.
  - More constructs may be added to \( \mathcal{F} \) to allow the specification of labelling strategies. Given a function problem from a given source set into another given source set, one might think of labelling strategies for an \( \mathcal{F} \) model as strategies that decide which element of the source set needs to be assigned next and which element of the target set to try first as an image for the chosen element. Then these labelling strategies in an \( \mathcal{F} \) model will be converted into appropriate labelling strategies on the alternate \( \mathcal{L} \) models.
  - Some of the function problems have symmetry because either the elements of the source set are indistinguishable or the elements of the target set are indistinguishable, or both. A construct like \textit{indistinguishable}(S) may be added to \( \mathcal{F} \) to specify that the elements of set \( S \) are indistinguishable. In the presence of such a construct, the generation of the \( \mathcal{L} \) models may add appropriate symmetry-breaking constraints. For instance, when both the source and the target set of a function variable are indistinguishable, the rows and the columns of the \( \mathcal{L} \) that uses a 2d array of Booleans can be freely permuted leading to an exponential number of symmetric (non-)solutions. Lexicographically ordering the rows and lexicographically ordering the columns of such as array would reduce such symmetry [24].
  - There are some problems that are not function problems. For instance, consider the problem of assigning papers to reviewers. Each paper is assigned to many reviewers and each reviewer reviews many papers. In such a case, our desired output is best captured by a relation [22]. Extending the \( \mathcal{F} \) language to also allow relation variables can be done in a straightforward manner. There are at least three basic ways of representing a relation variable in terms of integer and set variables. The first representation is the same as the representation for a function variable that uses a 2d Boolean array, except that the constraint that each row-sum is 1 is dropped. The second alternative is to have a set variable for each element in the source set storing all the elements in the target set that are related to this element and similarly, in the third alternative, we may introduce a set variable for each element in the target set storing all the elements in the source set that it relates to. Combining these representations can be done in similar ways as we did for total functions except that we need to use different channelling constraints. Channelling between the first and the second representation of a relation variable can be achieved by making sure that the Boolean variable denoting the element \( a \) of the source set and the element \( b \) of the target set is set to 1 if and only if \( b \) is an element of the set variable denoting \( a \). When the first and the third representations are combined, the channelling constraints are very similar to the previous case, where the Boolean variable denoting the element \( a \) of the source set and the
element $b$ of the target set is set to 1 if and only if $a$ is an element of the set variable denoting $b$. Channelling the second and the third representation is achieved by making sure that $a$ is an element of the set variable denoting $b$ if and only if $b$ is an element of the set variable denoting $a$. As for the relation operations, the only difference from function operations is that an element of the source set may be now related to more than one element of the target set. Other extensions may also include a composite function/relation variable where the desired output is best captured by a composite function or relation, a graph variable where the desired output is best captured as a graph, and a tree variable where the desired output is a tree, etc.

- **Global constraints for the channelling constraints**: In this thesis, we have shown that there might be a great gain in having combined models where the variables are linked through channelling constraints. Global constraints could be developed to efficiently maintain generalized arc-consistency on these channelling constraints.

- **More heuristics**: more heuristics may be developed for either choosing between different constraint formulations or choose between different models.

- **More theoretical analysis**: The theoretical analysis on models of injections may be extended to cover other local consistency properties as well as to include more models. The same analysis may be done for other types of problems.

- **Combining models for partial function problem**: integration of different representation of the partial function problem may be done, especially if we find good target applications.

- **Making $\mathcal{L}$ a solver-independent notation**: the $\mathcal{L}$ language can be further developed so as to become a solver-independent language. This will allow $\mathcal{L}$ to be widely used by different researchers using different CP languages to describe their models without using the specific syntax of the CP language they used. This will facilitate the reading and comparison of different models implemented by different researchers in different CP languages.

Finally, tools developed elsewhere such as the ones in [32, 4] can be added as extra components to our tool.
Bibliography


154


