IT Licentiate theses
2005-003

High-Level Modelling and Local Search

MAGNUS ÅGREN

UPPSALA UNIVERSITY
Department of Information Technology

UPPSALA
UNIVERSITET

High-Level Modelling and Local Search

BY
MAGNUS ÅGREN

August 2005

DIVISION OF COMPUTING SCIENCE
DEPARTMENT OF INFORMATION TECHNOLOGY
UPPSALA UNIVERSITY
UPPSALA
SWEDEN

Dissertation for the degree of Licentiate of Philosophy in Computer Science
at Uppsala University 2005

High-Level Modelling and Local Search

Magnus Ågren
agren@it.uu.se

Division of Computing Science
Department of Information Technology
Uppsala University
Box 337
SE-751 05 Uppsala
Sweden

http://www.it.uu.se

ISSN 1404-5117
Printed by the Department of Information Technology, Uppsala University, Sweden

How to Read this Thesis

This thesis comprises a cover chapter, three appended papers (Paper A, Paper B and, Paper C) and two appendices. The cover chapter features a general introduction to the area of interest, as well as an overview of each of the three papers and how these are related. The appendices correspond to Paper A and are referred to in its part of the cover chapter.

The interested reader should start with the abstract and the cover chapter. This should be enough to get an idea of which paper to read next. Each paper is then self-supporting and the reader could read them in any order of preference, even though the proposed one, A – B – C, may be the most natural one from our point of view.

Related work is discussed mainly in the respective papers, and only partly in the cover chapter.

Abstract

Combinatorial optimisation problems are ubiquitous in our society and appear in such varied guises as DNA sequencing, scheduling, configuration, airline-crew and nurse rostering, combinatorial auctions, vehicle routing, and financial portfolio design. Their efficient solution is crucial to many people and has been the target for much research during the last decades. One successful area of research for solving such problems is constraint programming. Yet, current-generation constraint programming languages are considered by many, especially in industry, to be too low-level, difficult, and large. In this thesis, we argue that solver-independent, high-level relational constraint modelling leads to a simpler and smaller language, to more concise, intuitive, and analysable models, as well as to more efficient and effective model formulation, maintenance, reformulation, and verification. All this can be achieved without sacrificing the possibility of efficient solving, so that even time-pressed modellers can be well assisted. Towards this, we propose the ESRA relational constraint modelling language, showcase its elegance on some real-life problems, and outline a compilation philosophy for such languages.

In order to compile high-level languages such as ESRA to current generation constraint programming languages, it is essential that as much support as possible is available in these languages. This is already the case in the constructive search area of constraint programming where, e.g., different kinds of domain variables, such as integer variables and set variables, and expressive global constraints are readily available. However, in the local search area of constraint programming, this is not yet the case and, until now, set variables were for example not available. This thesis introduces set variables and set constraints in the local search area of constraint programming and, by doing this, considerably improves the possibilities for using local search. This is true both for modelling and solving problems using constraint-based local search, as well as for using it as a possible target for the compilation of ESRA models. Indeed, many combinatorial optimisation problems have natural models based on set variables and set constraints, three of which are successfully solved in this thesis.

When a new set constraint is introduced in local search, much effort must be spent on the design and implementation of an appropriate incremental penalty function for the constraint. This thesis introduces a scheme that, from a high-level description of a set constraint in existential second-order logic with counting, automatically synthesises an incremental penalty function for that constraint. The performance of this scheme is demonstrated by solving real-life instances of a financial portfolio design problem that seem unsolvable in reasonable time by constructive search.

Acknowledgements

First of all I would like to thank my advisors Pierre Flener and Justin Pearson for their support during these first years of my PhD. Thank you for having faith in me and letting me explore a slightly different research path than perhaps was originally intended. Thank you for many interesting discussions, not only about research. I also appreciate very much the research trips we have made together to Brown University and to various conferences. I would also like to thank Pascal Van Hentenryck for hosting us during our visits to Brown.

Thanks also to the people in Singapore, especially Joxan Jaffar and Roland Yap, for giving the CS4210 course which is the reason for why I got interested in constraint programming in the first place. Thank you also Mats Carlsson and Nicolas Beldiceanu for the opportunity to do my Master's Thesis in constraint programming at SICS. This was probably a very good entrance ticket to my current PhD position.

I should also like to thank all my friends in Uppsala for making the years I have spent there very happy. Special thanks to Harald and Tomas for getting me interested in the more theoretical aspects of Computer Science. Thank you Jim for being a great officemate during these years, I am sure we still have lots of topics to sort out in 1410.

I would also like to thank my parents and my sister for their support, and finally, I want to thank Anna for her love and everlasting support during these years. Without you this thesis would never have been finished.

This research has been supported in part by grant 221-99-369 of VR, the Swedish Research Council, by institutional grant IG2001-67 of STINT, the Swedish Foundation for International Cooperation in Research and Higher Education, as well as by Project C/1.246/HQ/JC/04 and its successor of EuroControl, the European Organisation for the Safety of Air Navigation.

List of Appended Papers

[**Paper A**] P. Flener, J. Pearson, and M. Ågren. *Introducing* ESRA*, a Relational Language for Modelling Combinatorial Problems.* In: M. Bruynooghe (editor), Revised selected papers of the 13th International Symposium on Logic Based Program Synthesis and Transformation - LOPSTR 2003, pp. 214–232, Lecture Notes in Computer Science, volume 3018. Springer-Verlag, 2004.

[**Paper B**] M. Ågren, P. Flener, and J. Pearson. *Set Variables and Local Search.* In: R. Barták and M. Milano (editors), Proceedings of the 2nd International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - CP-AI-OR 2005, pp. 19–33, Lecture Notes in Computer Science, volume 3524, Springer-Verlag, 2005.

[**Paper C**] M. Ågren, P. Flener, and J. Pearson. *Incremental Algorithms for Local Search from Existential Second-Order Logic.* In: P. van Beek (editor), Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming - CP 2005, Lecture Notes in Computer Science, Springer-Verlag, 2005.

Comments on My Participation

[**Paper A**] I took part in the discussions and in developing the ideas behind the paper. I came up with the grammar in Figure 1 and wrote part of Section 2.3. I also wrote the corresponding appendices in this thesis and designed and implemented a parser for ESRA.

[**Paper B**] I am the principal author of this paper. I wrote most parts of it except for Section 3.3 where I contributed to the beginning including Example 5 and Algorithm 1. I also implemented the local-search framework and the applications presented in the paper.

[**Paper C**] I took part in the discussions and in developing the ideas behind the paper. I wrote Sections 3 – 5, 7.3, and 7.4 of the paper and contributed to the other sections. I also did all the implementation work for the paper except for the complete-search part of the application in Section 7.

Contents

1 Introduction

Combinatorial (optimisation) problems are ubiquitous in our society and appear in such varied guises as DNA sequencing, scheduling, configuration, airline-crew and nurse rostering, combinatorial auctions, vehicle routing, and financial portfolio design. Their efficient solution is crucial to many people and has been the target for much research during the last decades.

An archetypical example of a *combinatorial problem* is the *World Map Colouring Problem*: Given is a map of the world, a set of colours, and a positive number k. The problem is to determine if it is possible to paint each country in one of the colours such that at most k colours are used and such that no two adjacent countries have the same colour.

If we limit ourselves to the *Nordic Countries Map Colouring Problem* (NCMCP) (i.e., the map over the countries Denmark, Finland, Iceland, Norway and Sweden), let the set of colours be $\{blue, green, purple, red, yellow\}$, and let $k = 3$, we may present a *solution* to the problem by colouring Sweden and Iceland in yellow, Denmark and Finland in red, and Norway in blue, as shown in Figure 1.[1] This problem may also be stated as a *combinatorial op-*

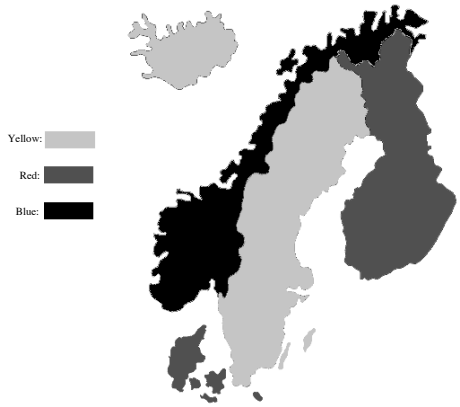


Figure 1: Coloured map of the Nordic countries.

timisation problem where, in addition to the conditions above, k is unknown rather than given and must be the *least number of colours* needed to colour the map. In our example, $k = 3$ *is* the least number of colours needed and, hence, Figure 1 shows an *optimal solution*.

1.1 Constraint Programming

Constraint Programming (CP) [15, 8, 1] is a framework for modelling and solving combinatorial problems. It is based on the fact that a large and complex problem may be represented by a set of (often high-level) constraints.

[1] Sweden and Denmark are considered adjacent due to the Öresund bridge.

These constraints are in turn represented by efficient software components that, when they are combined with an appropriate search procedure, define an algorithm for solving the problem.

In the framework of CP, combinatorial problems and combinatorial optimisation problems are represented by *Constraint Satisfaction Problems* (CSPs) and *Constraint Optimisation Problems* (COPs) respectively.

Definition 1 *A* CSP *is a three-tuple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *where* $\mathcal{X} = \{x_1, \ldots, x_n\}$ *is a set of variables,* $\mathcal{D} = \{D_{x_1}, \ldots, D_{x_n}\}$ *is a set of domains, each* D_{x_i} *containing the set of possible values for the corresponding variable* x_i*, and* $\mathcal{C} = \{c_1, \ldots, c_m\}$ *is a set of constraints, each* c_i *being defined on a subset of* $\mathcal{X}$ *and specifying the valid combinations of values for those variables.*

Definition 2 *Let* $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *be a CSP and let* p^c *denote the number of variables of a constraint* $c \in \mathcal{C}$*. An* assignment for P *is a function* $k : \mathcal{X}' \longrightarrow \bigcup_{x \in \mathcal{X}'} D_x$*, where* $\mathcal{X}' \subseteq \mathcal{X}$*, with the condition that* $\forall x \in \mathcal{X}' : k(x) \in D_x$*. A* complete assignment for P *is an assignment* k *where* $domain(k) = \mathcal{X}$*. Now, an assignment* k *for* P *is* ***(i)*** *a* solution to *a constraint* $c(k(x_1), \ldots, k(x_{p^c})) \in \mathcal{C}$ *iff* $c(k(x_1), \ldots, k(x_{p^c}))$ *holds,* ***(ii)*** *a* partial solution *to* P *iff there is no* $c(x_1, \ldots, x_{p^c}) \in \mathcal{C}$ *such that* $\neg c(k(x_1), \ldots, k(x_{p^c}))$ *holds, and* ***(iii)****, a* solution to P *iff* $\forall c(x_1, \ldots, x_{p^c}) \in \mathcal{C} : c(k(x_1), \ldots, k(x_{p^c}))$ *holds.*

Example 1 Given Definition 1, the NCMCP can be formalised as a CSP $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ as follows. Let $\mathcal{X} = \{D, F, I, N, S\}$ be the set of variables, where D stands for the colour of Denmark, F stands for the colour of Finland, etc., and let $COLOURS = \{b, g, p, r, y\}$ be the common domain for all the variables, where b denotes the colour blue, g denotes the colour green, etc. Now, by letting $\neq$ be defined on the set $COLOURS$ such that for any $c, c' \in COLOURS$, $c \neq c'$ holds if and only if c and c' are not the same colour, the set of constraints $\mathcal{C} = \{D \neq S, F \neq N, F \neq S, N \neq S, |\{D, F, I, N, S\}| \leq 3\}$ correctly models the problem. Figure 1 corresponds to the *solution* k to P where $k(D) = r$, $k(F) = r$, $k(I) = y$, $k(N) = b$, and $k(S) = y$. Indeed, all of the disequalities are satisfied and $|\{k(D), k(F), k(I), k(N), k(S)\}| = |\{r, r, y, b, y\}| = |\{r, y, b\}| \leq 3$.

Definition 3 *A COP is a four-tuple* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, f \rangle$ *where* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *is a CSP and* $f : \mathcal{A} \rightarrow \mathbb{R}$ *is a function from the set of all complete assignments* $\mathcal{A}$ *for* $\langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ *to the real numbers whose value is to be minimised.*[2]

Example 2 Recall the CSP $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ of Example 1. The optimisation version of the NCMCP may be formalised as the COP $\langle \mathcal{X}, \mathcal{D}, \mathcal{C}, f \rangle$, where

[2]Without loss of generality we may restrict ourselves to minimisation COPs since a maximisation COP may be represented by a minimisation COP by considering the negated value of the value returned by its function.

f is the function defined as $f(k) = |\{k(D), k(F), k(I), k(N), k(S)\}|$. The value of f for the solution in Example 1 is 3, and this is indeed the minimal value.

The definitions above are very general and allow constraints to be stated on variables ranging over any kind of domains. The domains may for example be (subsets of) the set of integers, the set of reals, the set of booleans, a given set of values, or the power-set of a set of some type.

The most widely used domains in the CP community are probably finite subsets of the set of integers and much research has been devoted to the study of constraints and search procedures for such variables.

1.2 Constraint Programming and Constructive Search

Constraint programming research has historically been focused on *constructive search*, which means that the variables of the CSP (or COP) are assigned values from their domains in some systematic order until each variable has been assigned a value such that the constraints are satisfied or such that a non-solution proof is obtained. (Another way of saying this according to Definition 2 is that a partial solution to a CSP P is extended into a solution to P if possible.) This gives rise to a *search tree* and a possible search tree for Example 1 is shown in Figure 2 on the next page, where the solution $k = \{D \mapsto r,\ F \mapsto r,\ I \mapsto y,\ N \mapsto b,\ S \mapsto y\}$ is shown as the highlighted path.

Such a search tree may, even for small examples, become very large and exploring it entirely is not practical. In CP, this is remedied by removing branches of the search tree that can be shown not to contain a solution (or an optimal solution). This is done by the constraints of a CSP being *active entities* that, throughout the exploration of the search tree, remove values from the domains of the variables that cannot take part in a solution. As an example of this we consider the CSP $\langle\{x, y\}, \{D_x = 1 \ldots 4, D_y = 1 \ldots 3\}, \{x < y\}\rangle$, and notice that the constraint $<$ can be used to deduce that some of the values in the domains of x and y may be removed directly. Since the assigned value to x must be less than the assigned value to y, and y cannot be assigned a value larger than 3, any value in the domain of x larger than 2 cannot take part in a solution and may be removed. Similarly, since the assigned value to y must be larger than the assigned value to x, and x cannot be assigned a value smaller than 1, any value in the domain of y less than 2 may also be removed. Hence, the domains of x and y may immediately be shrunk to $D'_x = 1 \ldots 2$ and $D'_y = 2 \ldots 3$ respectively. Following the same reasoning after the assignment of a value to x implies further shrinking of the domain of y. The result this has to a corresponding search tree for the CSP is shown in Figure 3 on the next page, where the removed branches are shown dashed.

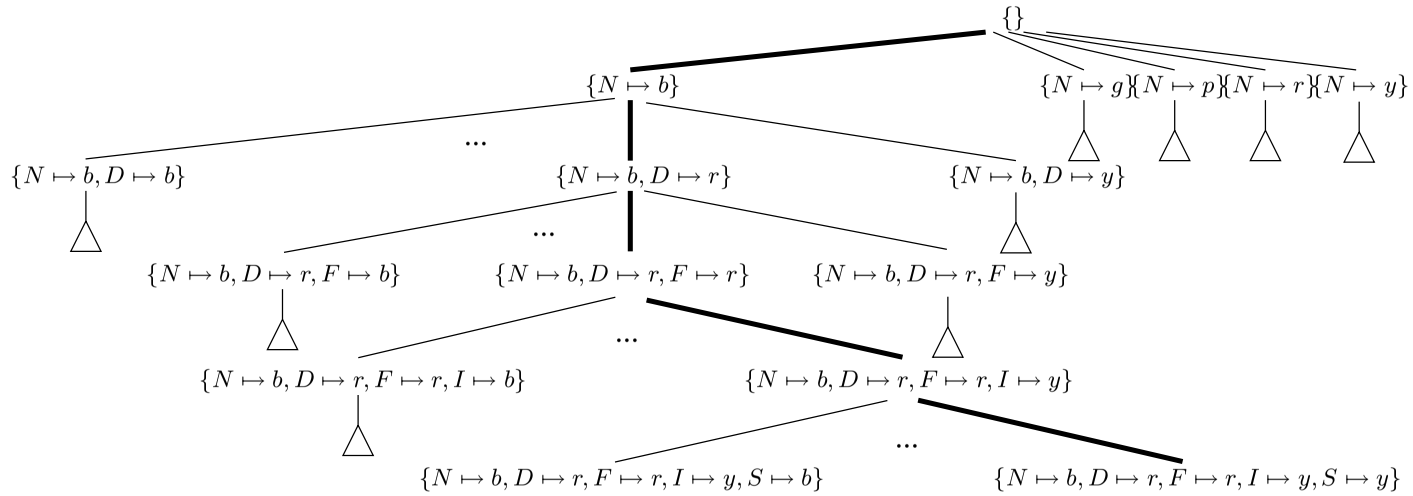


Figure 2: Search tree for the NCMCP.

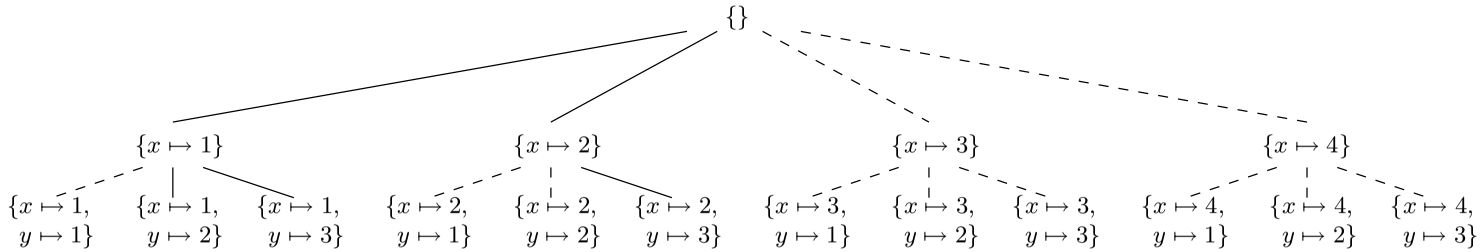


Figure 3: Search tree for the CSP $\langle \{x, y\}, \{D_x = 1 \dots 4, D_y = 1 \dots 3\}, \{x < y\} \rangle$.

While the examples so far include only very simple *binary constraints*, in general, constraint programming relies heavily on the existence of much more expressive and complex constraints. These constraints are referred to as *global constraints* and the standard example of such a constraint in CP is the *AllDifferent*$(\{x_1, \ldots, x_n\})$ constraint [14]. This constraint may be decomposed into the semantically equivalent logical formula $\forall i < j \in \{1, \ldots, n\} : x_i \neq x_j$, i.e., a set of $n(n-1)/2$ disequality constraints. The differences (and advantages) of the *AllDifferent* constraint are the space needed to represent it ($\mathcal{O}(n)$ as opposed to $\mathcal{O}(n^2)$) and the preserved information that n variables are to take distinct values. Indeed, if the $n(n-1)/2$ disequalities are used, we view each disequality in isolation and do not know anything about the others. If we instead use the *AllDifferent* constraint, we view all disequalities together and may take advantage of this when we design a *filtering algorithm* for the constraint.[3] The following example illustrates this.

Example 3 Consider the CSP $P = \langle\{x, y, z\}, \{D_x = \{1, 2\}, D_y = \{1, 2\}, D_z = \{1, 2, 5\}\}, \mathcal{C}\rangle$, and first assume that $\mathcal{C}$ is the set of constraints $\{x \neq y, x \neq z, y \neq z\}$. Given one of these constraints, say $x \neq y$, the only condition that allows us to remove a value from D_x (respectively D_y) is if D_y (respectively D_x) contains only one value. Hence, given the initial domains of P, nothing may be done to the domains of its variables.

Assume now that $\mathcal{C}$ is equal to $\{\mathit{AllDifferent}(\{x, y, z\})\}$. This allows us to reason on the domains of all variables at the same time. By doing this we may deduce that since $D_x = D_y = \{1, 2\}$, the values 1 and 2 must be reserved for x and y in any solution to the constraint. These values may therefore be removed from D_z, resulting in the domain $D'_z = \{5\}$ for z.

It should be clear that if global constraints are used instead of their decomposed counterparts, the resulting search tree may be smaller since more branches of the tree may be removed at a more shallow level.

1.3 Constraint Programming and Local Search

As discussed in Section 1.2, constructive search procedures for solving CSPs and COPs implicitly explore the complete search space by, for any given subtree, either traversing it completely, or proving that it cannot contain an (optimal) solution and, hence, that traversing it is unnecessary. In contrast, *local search* procedures only explore parts of the search space. This is done by an iterative procedure starting from a complete assignment that need not be a solution. Complete assignments in local search are usually referred to as *configurations* and we will use this terminology from now on. Each

[3] A filtering algorithm for a constraint is the algorithm that is used to remove values from the variables of the constraint, and to detect when the constraint cannot be satisfied.

iteration implies evaluating a set of configurations K that are very similar to the current one; the change between the current configuration k and an element k' in K may for example be that the value of a single variable differs. The set K is called the *neighbourhood* of k and in the search, a suitable candidate in K is picked as the current configuration in the next iteration. The iterative procedure stops when a sufficiently good configuration has been reached, or when some allocated resources have been exhausted.[4] It should be noted here that while a constructive search procedure is determined to find an (optimal) solution, no such guarantees can be made by a local search procedure. Completeness is traded for a usually more efficient algorithm.

Example 4 Recall the CSP corresponding to the NCMCP in Example 1. A sequence of three configurations for this CSP is shown in Figure 4, corresponding to an initial configuration and the picked neighbouring configurations in two consecutive iterations. The initial configuration is in this case $k = \{D \mapsto y, F \mapsto r, I \mapsto y, N \mapsto r, S \mapsto y\}$. In the first iteration, the colour of Norway is changed from red to blue resulting in the configuration k', and in the second one, the value of Denmark is changed from yellow to red resulting in the configuration k''. As can be seen, k'' corresponds to the solution in Example 1.

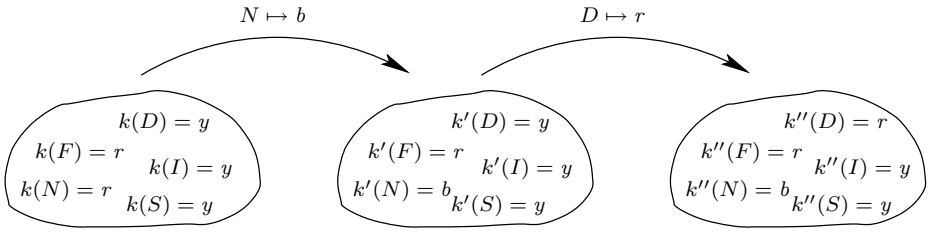


Figure 4: A sequence of three configurations for the NCMCP.

As opposed to constructive search, where the constraints of the CSP are used to *remove* values from the domains of their variables, in local search, the constraints are used to *guide* the search procedure in the right direction. For example, in the example above, how did we know that changing the values of N and D to b and r respectively would imply a configuration that represents a solution to the CSP? This is usually done by associating a *penalty function* to each constraint c of a CSP that maps a given configuration k to a numerical value $penalty(c)(k)$. This numerical value is the *penalty* of c with respect to k and is a measure on how violated c is. As a consequence, $penalty(c)(k)$ must be 0 if and only if c is satisfied with respect to k.

[4]This may for example mean that a maximum number of iterations has been reached, or that a timeout has occurred.

Let us now look at an example of this by considering the constraint $x \neq y$. A penalty function for this constraint may be defined as:

$$\mathit{penalty}(x \neq y)(k) = \begin{cases} 1, \text{ if } k(x) = k(y) \\ 0, \text{ otherwise} \end{cases} \tag{1}$$

Hence, given a configuration k, the penalty of the constraint $x \neq y$ is 1 if the values of x and y with respect to k are the same, and 0 if they are different.

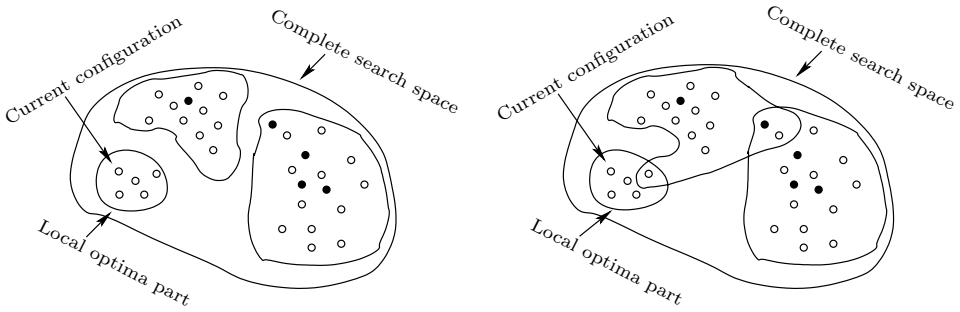
Now, given a CSP $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ and a configuration k for P, the *penalty* of P with respect to k is the sum of the penalties of the constraints in $\mathcal{C}$.

Example 5 Recall the CSP $P = \langle \{x, y, z\}, \{D_x = \{1, 2\}, D_y = \{1, 2\}, D_z = \{1, 2, 5\}\}, \mathcal{C} \rangle$ in Example 3, assume that $k = \{x \mapsto 1, y \mapsto 1, z \mapsto 1\}$ is the initial configuration for P, and that $\mathcal{C} = \{x \neq y, x \neq z, y \neq z\}$. The penalty of each of the constraints in $\mathcal{C}$ is 1, since they are all violated, hence the penalty of P is $1 + 1 + 1 = 3$.

While for the $\neq$-constraint it is simple to define a penalty function, to define penalty functions for *global constraints* is more complicated. In any case, it is very important that different constraints are given *balanced* penalty functions that are *comparable* to each other, and that *naturally reflect* how much violated their respective constraints are. No constraint should be easier (or harder) in general to satisfy compared to the others, and no constraint should give a too low or a too high penalty with respect to a configuration. If this is not the case, it will be harder for the local search procedure to be guided in the right direction. It may for example be impossible to escape parts of the search space that only contain local optima, as illustrated in Figure 5 on the next page, where filled circles represent solution configurations and non-filled circles represent non-solution configurations. In the figure, (a) and (b) show views of the search space for a CSP with the assumption that the penalty functions of the constraints are unbalanced and balanced respectively. In both cases, the search procedure currently explores a part of the search space in which only local optima exists. In (a), since the penalty functions of the constraints are unbalanced, the search procedure will never reach a part of the search space in which there is a solution. However, the balanced penalty functions for the constraints in (b) will guide the search procedure in the right direction, as shown by the overlapping parts of the search space.

Example 6 Recall once again the CSP P in Example 3, assume that $k = \{x \mapsto 1, y \mapsto 1, z \mapsto 1\}$ is the initial configuration for P, and that $\mathcal{C} = \{\mathit{AllDifferent}(\{x, y, z\})\}$. A penalty function for the *AllDifferent* constraint may be defined as:

$$\mathit{penalty}(\mathit{AllDifferent}(\mathcal{X}))(k) = |\mathcal{X}| - \left| \bigcup_{x \in \mathcal{X}} k(x) \right| \tag{2}$$



(a) (b)

Figure 5: Examples of search spaces where the corresponding constraints are unbalanced (a) and balanced (b) respectively.

Hence, given a configuration k, the penalty of the *AllDifferent* constraint is the number of *repeated values* in $\mathcal{X}$ with respect to k.[5] This definition corresponds to what is denoted variable-based violation cost in the soft-constraints area (see for example [12]). Another variant is to count the number of equal pairs in $\mathcal{X}$ with respect to k, as is done in [5], for example.

The penalty of $\mathit{AllDifferent}(\{x, y, z\})$ according to (2) with respect to k is $\mathit{penalty}(\mathit{AllDifferent}(\{x, y, z\}))(k) = 2$, since there are two repeated values (for example $k(x) = 1$ and $k(y) = 1$) with respect to k. This may be interpreted in the way that at least two variables must change in order to satisfy the constraint. If we change the value of any of the variables in $\{x, y, z\}$, the penalty will decrease by 1. Hence, we obtain *feedback* that any such change is a step closer a satisfied constraint.

Another penalty function for the *AllDifferent* constraint may be defined as (let $\mathcal{X}$ denote the set $\{x_1, \ldots, x_n\}$):

$$\mathit{penalty}(\mathit{AllDifferent}(\mathcal{X}))(k) = \begin{cases} 1, & \text{if } \exists i \neq j : k(x_i) = k(x_j) \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Hence, given a configuration k, the penalty of the *AllDifferent* constraint is 1 if there exists *at least one* distinct pair of variables x_i and x_j such that $k(x_i) = k(x_j)$, and 0 otherwise.

This penalty function is not very useful since it reports a very small (and constant) penalty of the constraint given any violating configuration. The only cases for which it will be able to guide the search are the ones where k

[5]The number of repeated values in $\mathcal{X}$ with respect to k may be obtained by, until $\mathcal{X} = \emptyset$, iteratively removing a variable x in $\mathcal{X}$, and placing the resulting value $k(x)$ in an initially empty set $\mathcal{X}_k$. If, for a given $x \in \mathcal{X}$, the value $k(x)$ is already in $\mathcal{X}_k$, a counter r (initially zero) is increased by one. When this iterative procedure has finished, the value of r is the number of repeated values in $\mathcal{X}$ with respect to k.

is a configuration such that there exists *exactly one* distinct pair of variables x_i and x_j such that $k(x_i) = k(x_j)$. For example, for the single constraint in P and the configuration k above, the value of $\mathit{penalty}(\mathit{AllDifferent}(\{x, y, z\}))(k)$ according to (3) is 1. No matter what change we do to one of the variables in $\{x, y, z\}$ the value will still be 1, hence, in this case, we get *no feedback* from the penalty function.

In a real-life local search application, the number of iterations needed to find a (good enough) solution may be as large as 500,000. In each iteration a neighbourhood of size 1000 or larger is not uncommon. Due to this, it is impractical to recalculate from scratch the penalties of the constraints for each configuration when evaluating a particular neighbourhood. This is especially true for highly expressive and complex global constraints. One solution to this problem is to use *incremental algorithms* for calculating the penalties. This means that for a given constraint c, a current configuration k, and the penalty p_k of c with respect to k, the penalty of c with respect to a member k' in the neighbourhood of k is obtained by using the information of p_k and the *difference* between k and k'. There are different ways of doing this, e.g., by specialised algorithms for each constraint, by using *invariants* as in [16], or by reasoning about the constraints in terms of their *graph properties* as in [3].

2 High-Level Modelling

Current constraint programming languages are considered by many people to be too low-level and complex to use; there are many techniques a novice must learn to master before he or she becomes productive. Due to this, CP techniques are not as widely spread as they could (and ought to) be, especially in the industrial applications area. Users in general and novice users in particular would be much helped if they could be presented with a CP language that lets them focus more on what is important, i.e., *how to specify the problem* instead of *how to solve the problem*.

Paper A presents the language ESRA for specifying CSPs and COPs. The language is a conservative extension of second-order logic and features *existentially quantified relation variables* (and thus set variables) in addition to the traditional scalar variables. The language has intentionally been kept small with orthogonal constructs sufficient for modelling a large number of combinatorial optimisation problems.

More specifically, the main *contributions* of Paper A are the following:

- It introduces the *high-level language* ESRA, based on a first-order relational calculus, for the *elegant modelling* of combinatorial optimisation problems. This is illustrated by showing ESRA models of three real-life problems.

- It argues that the design of ESRA takes into account that *no low-level representation details are assumed* in models therein. Hence, the options for making such assumptions are left for the compilation phase.

- It argues that ESRA-models *can be compiled into efficient models* in lower-level (constraint programming) languages of today.

In addition to Paper A, this thesis also presents the grammar and denotational semantics of ESRA in Appendices A and B respectively.

In order to get an idea about the language, let us start by giving an ESRA model of a real-life application: the *Social Golfer Problem* (SGP).

The SGP is stated as follows: In a golf club, there are N players, each of whom plays golf once a week (constraint c_1) and always in G groups of size S (c_2), i.e., $N = G \cdot S$. The objective is to determine whether there is a schedule of W weeks of play for these golfers, such that there is at most one week where any two distinct players are scheduled to play in the same group (c_3). An implied constraint is that every group occurs exactly $S \cdot W$ times across the schedule (c_4). See Problem 10 at `http://www.csplib.org` for more information. A solution to the SGP may be thought of as a function *Schedule* from the set of golfers and the set of weeks to the set of groups, that satisfies the constraints $c_1 - c_4$ above, i.e., a function that maps a golfer g and a week w to the group $Schedule(g, w)$ that g plays in in week w.

A model of the SGP in ESRA is shown in Figure 6 and displays much of the capabilities of the language, as explained next. An ESRA model starts

```
(1)  cst G, S, W : ℕ
(2)  dom players = 1...G * S,  weeks = 1...W,  groups = 1...G
(3)  var Schedule : (players × weeks) ⟶^{S*W} groups
(4)  solve
(5)      ∀(g : groups, w : weeks)
(6)          count(S)(p : players | Schedule(p, w) = g)
(7)      ∧
(8)      ∀(p1 < p2 : players)
(9)          count(0...1)(w : weeks | Schedule(p1, w) = Schedule(p2, w))
```

Figure 6: An ESRA model of the SGP.

with the declaration of constants, domains, and variables. This is exemplified in lines (1) and (2) in Figure 6, where the constants G, S, and W are defined as (yet unassigned) natural numbers, and the domains *players*, *weeks*, and *groups* are defined as integer ranges (seen as sets) with respect

to the constants.[6] The variable *Schedule* is declared in line (3), ranging over the set of all functions from the set of all pairs in the cross product of *players* and *weeks* (constructed by the binary infix operator $\times$) to the set *groups*. The operator $\longrightarrow^{S*W}$ reveals one of the useful constructs in ESRA available to state clear and concise problem models: the expression $S * W$ declares that *Schedule* is a function such that every element in *groups* is mapped to by exactly $S \cdot W$ (player, week) pairs. Hence, the constraint c_4 is stated already in the declaration of the variable, as well as the constraint c_1, because of the totality of the function.

After the declarations comes the objective, in which a set of constraints is to be solved and possibly also such that a numerical expression is optimised. In the SGP case, the variable *Schedule* should be further constrained with respect to the constraints c_2 and c_3. This is done in lines (5) – (6) and (8) – (9) respectively. The constraint c_2 is stated by a universal quantification in which each group in each week is constrained to be of size S by a count constraint. The expression $(p : players|Schedule(p, w) = g)$ (from now on referred to as E) denotes the set of all players scheduled to play in week w and group g, and the formula $\text{count}(S)E$ constrains the cardinality of E to be equal to S. In this formula, S is actually a shorthand for the singleton set $\{S\}$ and in general, any ground set T of natural numbers may take its place with the semantics that the cardinality of E must be in T. The constraint c_3 is stated using a similar formula for each distinct pair of golfers.

In the model above (and generally in ESRA models), we did not commit ourselves to using any specific data structures for the variable and the constraints. This is good because then we are allowed to make (automatically) such decisions during the compilation/solving phase. We could for example try to design a solver that works directly on the relation variables such as the function variable *Schedule*. Another and currently more realistic option is to transform models in ESRA into models in an existing constraint programming language, and then solve those models using the solvers currently at hand. If this is done there are many issues that arise, there are for example different ways one may represent a relation variable. If *set variables* exist in the target language, one may for example represent a relation variable $R : A \times B$ by the set variable $S \in 2^{U_S}$, where $U_S = A \times B$, such that (a, b) are related in R if and only if $(a, b) \in S$. If only integer variables exist in the target language, one may represent R by a two-dimensional matrix M indexed by the elements in A and B, such that (a, b) are related in R if and only if $M[a][b] = 1$. Depending on the variable representation chosen, there are then different options for stating the constraints of the ESRA model.

In addition to the chosen representation for variables and constraints, which actual *solving algorithm* to use is of course also open. We may for

[6]Note that constants (and domains) may be initialised by a corresponding data file or interactively at runtime, e.g., as in Figure 6 for the constants G, S, and W.

example use constructive search, local search, or any combination thereof. Currently in constraint programming, there exist many options for using constructive search regarding solvers with different kinds of domain variables and constraints, and we may take advantage of this when translating ESRA models. However, these options are more limited for local search and, hence, Papers B and C are an answer to this by providing higher-level local search possibilities based on set variables and set constraints.

3 High-Level Solving with Local Search

Being historically mostly focused on constructive search, the CP community has recently started to bring its ideas also to local search. This implies a compositional strategy with high-level constructs such as global constraints and the separation of modelling from search (see for example [20, 16, 11, 5, 4, 9, 17, 3]).

Even though this research has come far, witnessed by for example the possibilities of the COMET programming language and system [9, 17, 19, 18], the variables in local search have up until now been restricted to those that are assigned scalar values, such as *integer variables*. This contrasts the constructive search area of CP, where, e.g., *set variables* have been around for a long time, i.e., variables that are assigned sets of values [6, 10, 13, 2]. Being able to use set variables and set constraints in CP (whether it is in conjunction with constructive search or with local search) clearly implies a *modelling advantage* for applications that have natural set-based models. It may also imply a *solving advantage* since it allows more structural information from the model to be preserved at the solving level.

Since one of the ideas behind the ESRA language is to translate such models automatically into (different) models of current CP languages, it is crucial that as much support as possible in terms of available domains and constraints is available in the target languages. This will make it easier to translate the ESRA models as well as provide more options when it comes to providing *different* target models. Papers B and C are steps in this direction by providing set variables and set constraints for local search, as well as automatic derivation of incremental algorithms from high-level specifications of set constraints.

3.1 Set Variables and Local Search

A *set variable* is a variable whose associated domain is a *power-set* of a set of values called its *universe*.

Definition 4 *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP. A variable $S \in \mathcal{X}$ is a* set variable *if its corresponding domain $D_S = 2^{U_S}$, where U_S is a finite set of values of some type, called the* universe *of S.*

Paper B *introduces set variables and set constraints in local search.* Recall that in a local search setting of CP, we deal with complete assignments known as configurations. In the search, we move between such configurations until we reach a satisfactory one. The constraints are used to guide the search, i.e., which neighbour of a current configuration to choose as the next one. In order to do this, each constraint is associated with a penalty function that expresses how far away the constraint is from being satisfied with respect to a given configuration. Also, as was mentioned in Section 1.3, it is very important that different constraints are associated with penalty functions that are *comparable* to each other (we say that such penalty functions are *balanced*) as well as naturally reflect how much violated the respective constraints are. Hence, in order to use set constraints in local search, we need to know how to define such penalty functions for those constraints. This is the main focus of Paper B which presents a generic scheme for doing this.

More specifically, the main *contributions* of Paper B are the following:

- *Set variables and set constraints* are introduced in the local search area of CP. To do this, local-search concepts such as penalties, configurations, and neighbourhoods are put into a *set-variable framework.*

- In order to be able to use set constraints generally in local search, it suggests a *generic scheme* for defining *balanced penalty functions* for set constraints. This scheme is then used to give the *penalty functions of five (global) set constraints.*

- In order to obtain efficient solution algorithms, Paper B proposes methods for the *incremental penalty maintenance* between configurations of the introduced set constraints.

- The concepts introduced in the paper are used to *model and solve two real-life problems* with good results.

One possible translation of the relational model of the SGP in Figure 6 into a set-based model according to the ideas in Paper B is shown in Figure 7. **Please note** that while the ESRA model in Figure 6 was given in (a pretty-printed version of) that language, the set-based model in Figure 7 is given in pseudo-code. The constraints used are actual entities in the solver for set variables and, in reality, the set-based model is coded in the host-language of the solver (currently OCaml [7]), where the set variables are stored in a $G \cdot W$ matrix and the constraints are stated on the rows and columns of that array by regular loops.

In the model of Figure 7, the identifiers G, S, W, *groups*, *weeks* and *players* are assumed to refer to the same things as in the ESRA model. The set of variables is the set $\{G_{(g,w)} \mid g \in \mathit{players} \wedge w \in \mathit{weeks}\}$, where $G_{(g,w)}$ denotes the set of players playing in group g in week w. Hence, there are

(1) $G, S, W : \mathbb{N}$
(2) $players = 1 \ldots G * S,\ weeks = 1 \ldots W,\ groups = 1 \ldots G$
(3) $\forall w \in weeks : Partition(\{G_{(g,w)} \mid g \in groups\}, players)$
(4) $\forall g \in groups : \forall w \in weeks : |G_{(g,w)}| = S$
(5) $MaxIntersect(\{G_{(g,w)} \mid g \in groups \wedge w \in weeks\}, 1)$

Figure 7: Set-based model of the SGP.

$G \cdot W$ such *set variables* compared to the *single function variable* in the ESRA model.

Line (3) states that every player must play in each week (constraint c_1 from the description of the SGP in Section 2). This is done by a *Partition* constraint on the variables of each week. Saying that these groups of players for each week must be a partition of all the players clearly achieves this. Line (4) states that every group must be of size s (constraint c_2 from the description) by a cardinality constraint on each set variable in the model. Line (5) states the meeting constraint (constraint c_3 from the description), i.e., that no two players should meet more than once across the schedule. This is done by a *MaxIntersect* constraint over all the variables with the meaning that the intersection of any two distinct variables (and hence groups) must be at most 1.

The constraints above now need associated penalty functions in order to be used in a local search setting. In order to do this in a general way, Paper B introduces the following definition in order to measure the *change* of a set variable.

Definition 5 *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP, let k be a configuration for P, and let $S \in \mathcal{X}$. An* atomic set operation on $k(S)$ *is one of the following changes to $k(S)$:*

1. *Add a value d to $k(S)$ from its complement, denoted $add(k(S), d)$.*
2. *Remove a value d from $k(S)$, denoted $remove(k(S), d)$.*

When we define the penalty function of a constraint, we usually reason about performing (shortest) sequences of atomic set operations and what effect that will have on a given variable.

Example 7 Assume that S and S' are set variables and that k is a configuration such that $k(S) = \{a, b, c\}$ and $k(S') = \emptyset$. Performing the sequence of set operations $\Delta = [add(k(S), d),\ remove(k(S), b),\ add(k(S'), b)]$ on $k(S)$ and $k(S')$ will yield $\Delta(k(S)) = \{a, c, d\}$ and $\Delta(k(S')) = \{b\}$ respectively.

Definition 5 may now be used in order to give a measure on how expensive it is to change the values of a set of set variables X with respect to a configuration k into the values of X with respect to another configuration k' such that some constraint c defined on X is satisfied. This is done by counting the *least number of* atomic set operations necessary to satisfy c, as shown in the next definition.

Definition 6 *Let $P = \langle \mathcal{X}, \mathcal{D}, \mathcal{C} \rangle$ be a CSP and let $\mathcal{A}$ be the set of all configurations for P. Let $c \in \mathcal{C}$ be a constraint defined on a set of set variables $X \subseteq \mathcal{X}$. The* penalty of c, *penalty$(c) : \mathcal{A} \rightarrow \mathbb{N}$, is the length of a shortest sequence of atomic set operations that must be performed in order to satisfy c with respect to a given configuration $k \in \mathcal{A}$.*

From this definition it follows that $\mathit{penalty}(c)(k) = 0$ if and only if c is satisfied with respect to k. To come up with penalty functions that comply with this definition is not always easy, as can be seen in Paper B for the *MaxIntersect* constraint.

As an example, we will look at the penalty function for the $\mathit{Partition}(X, q)$ constraint. This constraint is satisfied with respect to a configuration k if and only if the set $X = \{S_1, \ldots, S_n\}$ is a partition of q with respect to k, where q is a ground set, i.e., if and only if the following formula holds:

$$\forall i < j \in 1 \ldots n : k(S_i) \cap k(S_j) = \emptyset \wedge \bigcup_{S \in X} k(S) = q \tag{4}$$

Now, the penalty of a $\mathit{Partition}(X, q)$ constraint under k is equal to the length of a shortest sequence Δ of atomic set operations that must be performed in order for (4) to hold. The following penalty function expresses this:

$$\mathit{penalty}(\mathit{Partition}(X, q))(k) = \underbrace{\left(\sum_{S \in X} |k(S)|\right) - \left|\bigcup_{S \in X} k(S)\right|}_{(a)} + \underbrace{\left|q - \bigcup_{S \in X} k(S)\right| + \left|\bigcup_{S \in X} k(S) - q\right|}_{(b)}$$

Indeed, we need to remove all repeated occurrences of any value in the partition for those variables to be all disjoint, and their number equals the difference between the sum of the set sizes and the size of their union (part (a)). Also, in order for the union of the variables in the partition to be equal to q, we need to add all unused elements of the set q to some set of the partition, as well as remove all elements in any of the sets in the partition that are not in q (part (b)).

Please note that the penalty function of the *Partition* constraint as defined in Paper B is incorrect since it does not take into account that we must remove all elements in any of the sets in the partition that are not

in the reference set. Hence the second term of the addition in (b) is not considered in Paper B.

3.2 Deriving Incremental Algorithms Automatically

As was mentioned in Section 1.3, the set constraints proposed in Paper B need incremental definitions of their penalty functions in order for a real-life problem to be solvable in reasonable time. While more elegant and general methods are possible, such as the ones used in [16, 9] and [3] for example, for those constraints, this was done in an ad hoc constraint-specific way. Assume now that we have a local-search system based on the constraints in Paper B (possibly extended with more constraints). If we come up with a new set constraint necessary for solving a particular problem that is not in our system yet, we will have to do (at least) the following two things before we may use it:

1. Come up with a penalty definition for the constraint, ideally one that complies with Definition 6.

2. Come up with an incremental algorithm for maintaining the penalty of the constraint between neighbouring configurations.

This may be a time-consuming and error-prone task. Hence it would be nice if it was possible, in addition to the built-in constraints, to *model* new constraints in a way such that the resulting constraints automatically had natural and balanced penalty functions with associated incremental penalty maintenance algorithms. This is what Paper C proposes for set constraints by providing the possibility of modelling constraints in existential second-order logic with counting.

More specifically, the main *contributions* of Paper C are the following:

- It proposes the usage of existential second-order logic with counting (denoted $\exists\mathrm{SOL}^+$) as a *high-level modelling language* for (user-defined) constraints. It accommodates set variables and captures at least the complexity class NP.

- It proposes the design of a scheme for the *automated synthesis of incremental penalty calculation algorithms* from a description of a constraint in that language. An *implementation* of this scheme has been developed.

- A *new benchmark problem*, with applications in finance, for local search is proposed. Using the local search framework, *real-life instances* are solved exactly that seem unsolvable in reasonable time by constructive search.

As an example, assume that we do not have a built-in constraint for stating that S must be a strict subset of T, i.e., the constraint $S \subset T$. This may be expressed in $\exists\text{SOL}^+$ by the formula:

$$\exists S \exists T((\forall x(x \notin S \vee x \in T)) \wedge (\exists x(x \in T \wedge x \notin S))) \tag{5}$$

Similarly, a constraint stating that the size of the intersection between two set variables S and T can be at most m, i.e., the constraint $|S \cap T| \leq m$, may be modelled in $\exists\text{SOL}^+$ as follows:

$$\exists S \exists T \exists I((\forall x(x \in I \leftrightarrow x \in S \wedge x \in T)) \wedge |I| \leq m) \tag{6}$$

It should be noted here that a universal (or existential) quantification $\forall x\phi$ implies a quantification of x over the whole universe. It is also assumed that all set variables share the same universe (usually denoted $\mathcal{U}$). It is future work to introduce bounded quantification as well as individual initial domains. In (6), we introduce an additional set variable I and state that any value that is in both S and T must be in I and vice versa by a universal quantification over the elements in the universe. The primitive cardinality constraint may then be stated on I.

Actually, and as detailed in Paper C, we only consider a (complete) subset of the usual connectives of second order logic. We do this since it implies simpler expressions to define incremental algorithms for. In this smaller language, the formula (6) may be modelled as:

$$\exists S \exists T \exists I((\forall x((x \notin I \vee x \in S \wedge x \in T) \wedge (x \in I \vee x \notin S \vee x \notin T))) \wedge |I| \leq m) \tag{7}$$

Hence, (bi-)implication is removed from the language as well as negation. Any formula expressed using those connectives is easily expressed without them using standard transformation techniques. This is explained in more detail in Paper C.

Now, given a formula $\mathcal{F}$ in $\exists\text{SOL}^+$, the penalty function of $\mathcal{F}$ is defined inductively. For example, the penalty of a literal such as $x \in S$ (given a specific value for x) with respect to a configuration k is 1 if $x \notin k(S)$ and 0 otherwise. The penalty of the literal $|I| \leq m$ with respect to k is $\max(|k(I)| - m, 0)$, i.e., the difference between the actual cardinality $|k(I)|$ and the maximum m if $|k(I)| > m$, and 0 otherwise. The penalty of a conjunction $\phi \wedge \psi$ is the sum of the penalties of ϕ and ψ respectively. The penalty of a universal quantification $\forall x\phi$ is the sum of the penalties of ϕ with respect to every possible value for x. For disjunctions and existential quantifications the sum operators are replaced by min operators.

Example 8 Let $\mathcal{U} = \{a, b\}$ and let k be the configuration for $\{S, T\}$ such that $k(S) = k(T) = \{a\}$. The penalty of (5) with respect to k is 1, since there is one value that must be removed from S (or another value that must be added to T) in order for S to be a *strict subset* of T.

In order to incrementally evaluate the penalty function for a formula in $\exists\text{SOL}^+$ between different configurations, we represent the formulas in an extended form of syntax trees that we call *penalty trees*. The penalty tree of (5) is shown in Figure 8. The penalty for each subformula is stored in the

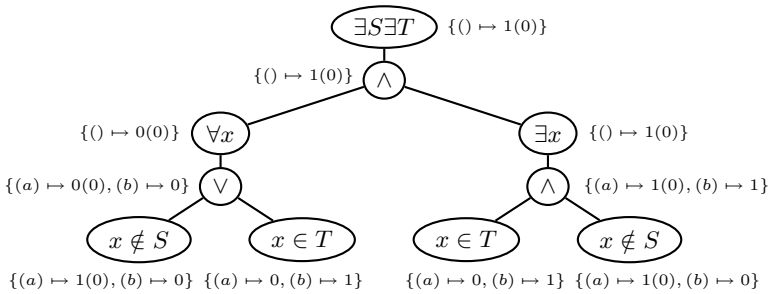


Figure 8: Penalty tree of (5).

corresponding subtree. Hence, the penalty for the whole formula is stored in the root node of the corresponding penalty tree. For the descendants of nodes representing subformulas that introduce bound variables, we must store the penalty with respect to *every* possible mapping of those variables. For example, the child node of a node for a subformula of the form $\forall x\phi$ will have a penalty stored for each $u \in \mathcal{U}$. This is illustrated in the penalty tree in Figure 8 in which the penalties with respect to the configuration k in Example 8 are shown (disregard the numbers in parentheses at this point). As can be seen, the penalty stored in the root node of the penalty tree is 1, corresponding to the penalty of (5) with respect to k.

Assume that we change k into k' such that the only difference between k and k' is that the value a has been removed from S. To update the penalty tree above (and hence to obtain the overall penalty with respect to k') we only need to consider the paths leading from any leaf containing S to the root node, and update the penalties for the nodes on those paths. The new penalties of the affected nodes are shown in parentheses in Figure 8. As can be seen, the penalty of (5) is now 0. Indeed, with respect to k', S *is* a strict subset of T.

Paper C presents three functions for initialising and incrementally maintaining penalty trees representing formulas in $\exists\text{SOL}^+$. These have been implemented in OCaml and make it possible to model (global) set constraints in $\exists\text{SOL}^+$ that may be used in the framework presented in Paper B.

4 Conclusion

In this thesis we present tools for simplifying and improving the modelling and solving of combinatorial (optimisation) problems. Although since long, much has been done in this area, there is still a long path to tread before

we reach the holy grail of computing: The user specifies the problem in his or her favourite language and the computer solves it.

The ESRA language is a step in this direction since it implies the specification of combinatorial problems at a very high level, thereby shifting the burden of designing efficient algorithms for a given problem from the user to the solver designer. In constraint programming, this has been already partly achieved by the existence of powerful global constraints for effectively shrinking the search space of an application. However, there are still many aspects to consider when modelling a given problem. For example, many problems require the user to deal with issues such as symmetry and implied constraints. This is far from trivial and requires lots of practice; even experts miss possible deductions. By using the ESRA language, expert knowledge, heuristics, and modelling tricks may be introduced in the solvers/compilers and thereby letting everybody take advantage of expert solving knowledge without having to know more about it than being able to specify their problems in that language.

In order to translate ESRA models into current constraint programming languages, it is important that these languages provide a broad range of constructs such that the translation process is not hindered. For example, having many global constraints at one's disposal is a good thing, since this makes it possible to produce more varied problem models with perhaps different levels of efficiency. Having different kinds of domain variables at one's disposal is also positive, due to similar reasoning.

As has been discussed in this thesis, the constructive search area of CP has traditionally been the more active area of research in the community. Many different solvers have been proposed and implemented for different kinds of domains, and much research has been performed in order to come up with efficient filtering algorithms for numerous global constraints. The local search area is still slightly behind in the number of different tools that are available and, until now, set variables and set constraints have not existed in local search solvers.

The introduction of set variables in local search may lead to more intuitive and simpler problem models, provides the user with a richer set of tools, and implies more preserved structure in underlying solving algorithms such as the incremental algorithms for maintaining penalties. Furthermore, the translation of ESRA models into set-based models may now be done without taking the actual search procedure into account. Hence, the impact of the chosen search procedure on translated ESRA models may now be compared also for set variables, a comparison having only been possible earlier for scalar variables.

When ESRA models are translated into set-based local search models, since the number of available constraints are still limited, we may have to come up with new constraints. As said in Section 3.2, this poses difficulties both in terms of defining the actual penalty function for the new constraint

and implementing an incremental algorithm for the same. The introduced scheme based on $\exists\text{SOL}^+$ is a solution to this problem. With this scheme it is possible to experiment with a modelled version of a new constraint before actually implementing a perhaps more efficient incremental algorithm for it compared to the one automatically derived by the system.

References

[1] Krzysztof R. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[2] Francisco Azevedo and Pedro Barahona. Applications of an extended set constraint solver. In *Proceedings of the ERCIM / CompulogNet Workshop on Constraints*, 2000.

[3] Markus Bohlin. *Design and Implementation of a Graph-Based Constraint Model for Local Search*. PhL thesis, Mälardalen University, 2004.

[4] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. In K. Steinhöfel, editor, *Proceedings of SAGA 2001, First International Symposium on Stochastic Algorithms: Foundations and Applications*, volume 2264 of *LNCS*, pages 73–90. Springer-Verlag, 2001.

[5] Philippe Galinier and Jin-Kao Hao. A general approach for constraint solving by local search. In *Proceedings of CP-AI-OR'00*, 2000.

[6] Carmen Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.

[7] Xavier Leroy *et al*. *The Objective Caml System release 3.08*. Institut National de Recherche en Informatique et en Automatique, jul 2004. Available at `http://caml.inria.fr`.

[8] Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. The MIT Press, 1998.

[9] Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):101–110, 2002. Proceedings of OOPSLA'02.

[10] Tobias Müller and Martin Müller. Finite set constraints in Oz. In François Bry, Burkhard Freitag, and Dietmar Seipel, editors, *Proceedings of 13th Workshop Logische Programmierung*, pages 104–115, Technische Universität München, sep 1997.

[11] Alexander Nareyek. Using global constraints for local search. In E.C. Freuder and R.J. Wallace, editors, *Constraint Programming and Large Scale Discrete Optimization*, volume 57 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 9–28. American Mathematical Society, 2001.

[12] T. Petit, J.-C. Régin, and C. Bessière. Specific filtering algorithms for over constrained problems. In Toby Walsh, editor, *Proceedings of CP'01*, volume 2293 of *LNCS*, pages 451–463. Springer-Verlag, 2001.

[13] Jean-François Puget. Finite set intervals. In *Proceedings of CP'96 Workshop on Set Constraints*, 1996.

[14] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 362–367. AAAI Press, 1994.

[15] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, 1989.

[16] Pascal Van Hentenryck and Laurent Michel. Localizer. *Constraints*, 5(1–2):43–84, 2000.

[17] Pascal Van Hentenryck and Laurent Michel. Control abstractions for local search. In Francesca Rossi, editor, *Proceedings of CP'03*, volume 2833 of *LNCS*, pages 65–80. Springer-Verlag, 2003.

[18] Pascal Van Hentenryck and Laurent Michel. Nondeterministic control for hybrid search. In Roman Barták and Michela Milano, editors, *Proceedings of CP-AI-OR'05*, volume 3524 of *LNCS*, pages 380–395. Springer-Verlag, 2005.

[19] Pascal Van Hentenryck, Laurent Michel, and Liyuan Liu. Constraint-based combinators for local search. In Mark Wallace, editor, *Proceedings of CP'04*, volume 3258 of *LNCS*, pages 47–61. Springer-Verlag, 2004.

[20] Joachim Paul Walser. *Integer Optimization by Local Search: A Domain-Independent Approach*, volume 1637 of *LNCS*. Springer-Verlag, 1999.

A Grammar of ESRA

Model

⟨Model⟩ ⟶ ⟨Decl⟩ ⟨Objective⟩

Declarations

⟨Decl⟩ ⟶ ⟨DomDecl⟩ | ⟨CstDecl⟩ | ⟨VarDecl⟩ | ⟨Decl⟩ ⟨Decl⟩

Domain Declarations

⟨DomDecl⟩ ⟶ `dom` ⟨Id⟩
| `dom` ⟨Id⟩ `=` ⟨Expr⟩

Constant Declarations

⟨CstDecl⟩ ⟶ `cst` ⟨Id⟩ `:` ⟨Expr⟩
| `cst` ⟨Id⟩ `=` ⟨Expr⟩ `:` ⟨Expr⟩

Variable Declarations

⟨VarDecl⟩ ⟶ `var` ⟨Id⟩ `:` ⟨Expr⟩

Objectives

⟨Objective⟩ ⟶ `solve` ⟨Expr⟩
| `minimise` ⟨Expr⟩ `such that` ⟨Expr⟩
| `maximise` ⟨Expr⟩ `such that` ⟨Expr⟩

Expressions

⟨Expr⟩ ⟶ ⟨Name⟩ | ⟨Appl⟩ | ⟨Tuple⟩ | ⟨NumExpr⟩
| ⟨SetExpr⟩ | ⟨Formula⟩

⟨Appl⟩ ⟶ ⟨Expr⟩ ⟨Expr⟩

⟨Tuple⟩ ⟶ `(` ⟨Exprs⟩ `)`

⟨Exprs⟩ ⟶ ⟨Expr⟩ | ⟨Expr⟩ `,` ⟨Exprs⟩

Numeric Expressions

⟨NumExpr⟩ ⟶ ⟨Int⟩
| `inf`
| `sup`
| ⟨Expr⟩ ⟨ArithBinOp⟩ ⟨Expr⟩
| ⟨ArithUnaryOp⟩ ⟨Expr⟩
| `card` ⟨Expr⟩
| `sum` `(` ⟨QuantExpr⟩ `)` `(` ⟨Expr⟩ `)`

⟨Int⟩ ⟶ ⟨Nat⟩ | `-`⟨Nat⟩

⟨Nat⟩ ⟶ ⟨Digit⟩ | ⟨Digit⟩ ⟨Nat⟩

⟨Digit⟩ ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Set Expressions

⟨SetExpr⟩ ⟶ `int` | `nat`
| `{}`
| `{` ⟨Exprs⟩ `}`
| ⟨Expr⟩ `..` ⟨Expr⟩
| `{` ⟨Expr⟩ `|` ⟨SvarDecls⟩ `}`
| `{` ⟨Expr⟩ `|` ⟨SvarDecls⟩ `|` ⟨Expr⟩ `}`
| ⟨Expr⟩ `[` ⟨Expr⟩ `]`
| ⟨Expr⟩ ⟨SetOp⟩ ⟨Expr⟩

⟨SvarDecls⟩ ⟶ ⟨LclVarDecl⟩
| ⟨LclVarDecl⟩ `/\` ⟨SvarDecls⟩

⟨SetOp⟩ ⟶ `#`
| `[` ⟨Expr⟩ `#` ⟨Expr⟩ `]`
| `[` ⟨Expr⟩ `#]`
| `[#` ⟨Expr⟩ `]`
| `[#]`
| `[->` ⟨Expr⟩ `]`
| `[->]`
| `->`
| `[+>` ⟨Expr⟩ `]`
| `[+>]`
| `+>`

Formulas

⟨Formula⟩ ⟶ `true` | `false`
| ⟨Expr⟩ ⟨BoolBinOp⟩ ⟨Expr⟩
| ⟨Expr⟩ ⟨RelOp⟩ ⟨Expr⟩
| `forall (` ⟨QuantExpr⟩ `)(` ⟨Expr⟩ `)`
| `exists (` ⟨QuantExpr⟩ `)`
| `count (` ⟨Expr⟩ `)(` ⟨QuantExpr⟩ `)`

⟨QuantExpr⟩ ⟶ ⟨QvarDecls⟩
| ⟨QvarDecls⟩ `|` ⟨Expr⟩

⟨QvarDecls⟩ ⟶ ⟨LclVarDecl⟩
| ⟨LclVarDecl⟩ `,` ⟨QvarDecls⟩

⟨LclVarDecl⟩ ⟶ ⟨Qvars⟩ : ⟨Expr⟩

⟨Qvars⟩ ⟶ ⟨Expr⟩ | ⟨Expr⟩ & ⟨Qvars⟩

Identifiers

⟨Name⟩ ⟶ ⟨Id⟩
| '⟨ASCII⟩'

⟨ASCII⟩ ⟶ *Any sequence of displayable ASCII characters.*

⟨Id⟩ ⟶ ⟨Letter⟩
| ⟨Letter⟩ ⟨DigitsLetters⟩

⟨Ids⟩ ⟶ ⟨Id⟩
| ⟨Id⟩ , ⟨Ids⟩

⟨Letter⟩ ⟶ A | ... | Z | a | ... | z

⟨DigitsLetters⟩ ⟶ (⟨Digit⟩ | ⟨Letter⟩ | _)
| (⟨Digit⟩ | ⟨Letter⟩ | _) ⟨DigitsLetters⟩

Operators

Relational Operators

⟨RelOp⟩ ⟶ < | =< | = | >= | > | !=

Arithmetic Operators

⟨ArithBinOp⟩ ⟶ + | - | * | / | %

⟨ArithUnaryOp⟩ ⟶ - | abs

Boolean Operators

⟨BoolBinOp⟩ ⟶ /\ | \/ | => | <= | <=>

B Denotational Semantics of ESRA

First of all, we define some basic sets such as the set $\mathbb{Z}$ of integers, the set $\mathbb{N}$ of natural numbers, the set $\mathbb{B}$ of booleans, and the set $\mathbb{A}$ of identifiers/names. We also define the set $\mathbb{T}$ of valid tuples, i.e., the set of tuples that may appear as atomic entities in the sets and relations of ESRA.

- $\mathbb{Z} \equiv \{\ldots, -1, 0, 1, \ldots\}$

- $\mathbb{N} \equiv \{0, 1, \ldots\}$
- $\mathbb{B} \equiv \{true, false\}$
- $\mathbb{A} \equiv [a-zA-Z\_]^+[a-zA-Z0-9\_]^* \cup$ '*ASCII**', where *ASCII* is any displayable ASCII character.
- $\mathbb{T} \equiv \bigcup_{n=0}^{\infty} (\mathbb{Z} \cup \mathbb{A})^n$

The grammar in Appendix A defines the syntactic domains of ESRA. We will here refer to those domains by using the names of their grammatical nonterminals. For example, *NumExpr* is the set of syntactically correct numerical expressions, *Formula* is the set of syntactically correct formulas, and so on.

In order to reason about decision variables, we introduce a set of universes *Universe* of bindings. For a given decision variable x in the domain of an element $\mathcal{U} \in$ *Universe*, $\mathcal{U}$ contains all possible bindings for x.

An *environment* Γ is a mapping from identifiers to values. The domain of an environment is always a subset of $\mathbb{A}$. We denote the *set of all environments* by *Environment*.

Model

A model consists of a sequence of domain-, constant-, and decision-variable declarations, followed by an objective. Evaluating a model means evaluating the objective under the environment defined by the domain- and constant declarations and the universe defined by the decision-variable declarations. We define the function $\mathcal{M}$ with the following signature:

$$\mathcal{M} : Model * Environment * Universe \longrightarrow 2^{Environment}$$

Hence, $\mathcal{M}$ takes as arguments an element of the syntactic domain *Model* (consisting of a declarations part and an objective part), an environment, and a universe, and returns a set of environments E. Each element in E contains bindings of all decision variables in the model with respect to the objective.

$$\mathcal{M}[\![\mathsf{d_1}\ \mathsf{d_2} \cdots\ \mathsf{d_m}\ \mathsf{v_1} \cdots\ \mathsf{v_n}\ \mathsf{o}]\!]^{\mathcal{U}}_{\Gamma} \equiv \mathcal{O}[\![\mathsf{o}]\!]^{\mathcal{U}_n}_{\Gamma_m}$$

where $\Gamma_1 = \mathcal{D}[\![\mathsf{d_1}]\!]_{\Gamma}$, $\Gamma_2 = \mathcal{D}[\![\mathsf{d_2}]\!]_{\Gamma_1}$, …, $\Gamma_m = \mathcal{D}[\![\mathsf{d_m}]\!]_{\Gamma_{m-1}}$, and $\mathcal{U}_1 = \mathcal{V}[\![\mathsf{v_1}]\!]^{\mathcal{U}}_{\Gamma_m}$, $\mathcal{U}_2 = \mathcal{V}[\![\mathsf{v_2}]\!]^{\mathcal{U}_1}_{\Gamma_m}$, …, $\mathcal{U}_n = \mathcal{V}[\![\mathsf{v_n}]\!]^{\mathcal{U}_{n-1}}_{\Gamma_m}$.

Domain- and Constant Declarations

Evaluating a domain- or constant declaration means adding a binding for an identifier to a given environment. We define the function $\mathcal{D}$ with the following signature:

$$\mathcal{D} \ : \mathit{DomDecl} \cup \mathit{CstDecl} * \mathit{Environment} \longrightarrow \mathit{Environment}$$

Hence, $\mathcal{D}$ takes as arguments an element d of $\mathit{DomDecl} \cup \mathit{CstDecl}$ and an environment Γ, and returns an environment Γ', where Γ' is an extension of Γ with the binding defined by d added.

$$\mathcal{D}[\![\texttt{dom } x \texttt{ = s}]\!]_\Gamma \ \equiv \ \{x \mapsto \mathcal{S}[\![\texttt{s}]\!]_\Gamma\} \cup \Gamma$$

$$\mathcal{D}[\![\texttt{cst } x \texttt{ = s1 : s2}]\!]_\Gamma \ \equiv \ \{x \mapsto \mathcal{S}[\![\texttt{s1}]\!]_\Gamma\} \cup \Gamma$$

Decision Variable Declarations

Evaluating a decision variable declaration means adding a binding for each possible value of the decision variable to a given universe. We define the function $\mathcal{V}$ with the following signature:

$$\mathcal{V} \ : \mathit{VarDecl} * \mathit{Environment} * \mathit{Universe} \longrightarrow \mathit{Universe}$$

Hence, $\mathcal{V}$ takes as arguments an element v of *VarDecl*, an environment, and a universe $\mathcal{U}$, and returns a universe $\mathcal{U}'$, where $\mathcal{U}'$ is an extension of $\mathcal{U}$ with the bindings defined by v added.

$$\mathcal{V}[\![\texttt{var } x \texttt{ : s}]\!]_\Gamma^{\mathcal{U}} \ \equiv \ \{x \mapsto s \mid s \in \mathcal{S}[\![\texttt{s}]\!]_\Gamma\} \cup \mathcal{U}$$

Objectives

Evaluating an objective means finding bindings to a set of decision variables such that a boolean formula is satisfied (and possibly such that a numerical expression is optimised). We define the function $\mathcal{O}$ with the following signature:

$$\mathcal{O} \ : \mathit{Objective} * \mathit{Environment} * \mathit{Universe} \longrightarrow 2^{\mathit{Environment}}$$

Hence, $\mathcal{O}$ takes as arguments an element o of *Objective*, an environment Γ, and a universe $\mathcal{U}$, and returns a set of extensions E of Γ, with decision variable bindings taken from $\mathcal{U}$, such that for each $\gamma \in E$, the boolean formula in o is satisfied (and possibly such that the numerical expression in o is optimised).

$$\mathcal{O}[\![\texttt{solve b}]\!]_\Gamma^{\mathcal{U}} \ \equiv \ \{\Gamma \cup \gamma \ \mid \gamma \subseteq \mathcal{U} \ \ \wedge \ \mathcal{E}[\![\texttt{b}]\!]_{\Gamma \cup \gamma}\}$$

$$\mathcal{O}[\![\texttt{minimise a such that b}]\!]^{\mathcal{U}}_{\Gamma} \equiv \{\Gamma \cup \gamma \mid \gamma \subseteq \mathcal{U} \wedge \mathcal{E}[\![\mathsf{b}]\!]_{\Gamma\cup\gamma} \wedge \forall\gamma' \subseteq \mathcal{U}\ (\mathcal{E}[\![\mathsf{b}]\!]_{\Gamma\cup\gamma'} \Rightarrow \mathcal{E}[\![\mathsf{a}]\!]_{\Gamma\cup\gamma} \leq \mathcal{E}[\![\mathsf{a}]\!]_{\Gamma\cup\gamma'})\}$$

$$\mathcal{O}[\![\texttt{maximise a such that b}]\!]^{\mathcal{U}}_{\Gamma} \equiv \{\Gamma \cup \gamma \mid \gamma \subseteq \mathcal{U} \wedge \mathcal{E}[\![\mathsf{b}]\!]_{\Gamma\cup\gamma} \wedge \forall\gamma' \subseteq \mathcal{U}\ (\mathcal{E}[\![\mathsf{b}]\!]_{\Gamma\cup\gamma'} \Rightarrow \mathcal{E}[\![\mathsf{a}]\!]_{\Gamma\cup\gamma} \geq \mathcal{E}[\![\mathsf{a}]\!]_{\Gamma\cup\gamma'})\}$$

Expressions

The meaning of evaluating an expression depends on its type. We define the function $\mathcal{E}$ with the following signature:

$$\mathcal{E} : \mathit{Expr} * \mathit{Environment} \longrightarrow \mathbb{T} \cup 2^{\mathbb{T}} \cup 2^{2^{\mathbb{T}}} \cup \mathbb{B}$$

Hence, $\mathcal{E}$ takes as arguments an element of the syntactic domain *Expr* and an environment, and returns, depending on the type of the expression, a tuple, a set of tuples, a set of sets of tuples, or a boolean value.[7]

$$\mathcal{E}[\![\mathsf{e}]\!]_{\Gamma} \equiv \begin{cases} \Gamma(\mathsf{e}) & \text{if } \mathsf{e} \in \mathit{Name} \text{ and } \mathsf{e} \in \text{domain}(\Gamma) \\ \mathcal{C}[\![\mathsf{e}]\!]_{\Gamma} & \text{if } \mathsf{e} \in \mathit{Name} \text{ and } \mathsf{e} \notin \text{domain}(\Gamma) \\ \mathcal{A}[\![\mathsf{e}]\!]_{\Gamma} & \text{if } \mathsf{e} \in \mathit{Appl} \\ \mathcal{T}[\![\mathsf{e}]\!]_{\Gamma} & \text{if } \mathsf{e} \in \mathit{Tuple} \\ \mathcal{N}[\![\mathsf{e}]\!]_{\Gamma} & \text{if } \mathsf{e} \in \mathit{NumExpr} \\ \mathcal{S}[\![\mathsf{e}]\!]_{\Gamma} & \text{if } \mathsf{e} \in \mathit{SetExpr} \\ \mathcal{F}[\![\mathsf{e}]\!]_{\Gamma} & \text{if } \mathsf{e} \in \mathit{Formula} \end{cases}$$

Names

$$\mathcal{C}[\![\mathsf{a}]\!]_{\Gamma} \equiv \mathsf{a}, \text{ for all } \mathsf{a} \in \mathbb{A}$$

Applications

The meaning of evaluating a function- or relation application depends on the type of the function/relation and of the argument. We define the function $\mathcal{A}$ with the following signature:

$$\mathcal{A} : \mathit{Appl} * \mathit{Environment} \longrightarrow \mathbb{T} \cup \mathbb{B}$$

Hence, $\mathcal{A}$ takes as arguments an element of *Appl* and an environment, and returns a tuple or a boolean value; a boolean value would be the result of a relation application.

[7] From this, one may believe that sets of sets are allowed as values in ESRA. However, some of the expressions (such as the ones in $2^{2^{\mathbb{T}}}$) are reserved for denoting *domains* of decision variables and, hence, the actual *values* of those variables would be *members* of the domains, e.g., a relation.

$$\mathcal{A}[\![\mathtt{e_1}\ \mathtt{e_2}]\!]_\Gamma \ \equiv\ \mathcal{E}[\![\mathtt{e_1}]\!]_\Gamma \mathcal{E}[\![\mathtt{e_2}]\!]_\Gamma$$

Tuples

The meaning of evaluating a tuple expression depends on the types of its members. We define the function $\mathcal{T}$ with the following signature:

$$\mathcal{T}\ : \mathit{Tuple} * \mathit{Environment} \longrightarrow \mathbb{T}$$

Hence, $\mathcal{T}$ takes as arguments an element of *Tuple* and an environment, and returns a tuple of evaluated values.

$$\mathcal{T}[\![\mathtt{(e_1,\ldots,e_n)}]\!]_\Gamma \ \equiv\ (\mathcal{E}[\![\mathtt{e_1}]\!]_\Gamma, \ldots, \mathcal{E}[\![\mathtt{e_n}]\!]_\Gamma)$$

Numeric Expressions

Evaluating a numeric expression means evaluating a numeric constant, or evaluating the argument(s) of some operator and to apply the operator to the resulting value(s). We define the function $\mathcal{N}$ with the following signature:

$$\mathcal{N}\ : \mathit{NumExpr} * \mathit{Environment} \longrightarrow \mathbb{Z}$$

Hence, $\mathcal{N}$ takes as arguments an element of the syntactic domain *NumExpr* and an environment, and returns an integer value.

$$\mathcal{N}[\![\mathtt{n}]\!]_\Gamma \ \equiv\ \mathtt{n}, \text{ for all } \mathtt{n} \in \mathbb{Z}$$

$$\mathcal{N}[\![\mathtt{inf}]\!]_\Gamma \ \equiv\ -\infty$$

$$\mathcal{N}[\![\mathtt{sup}]\!]_\Gamma \ \equiv\ \infty$$

$$\mathcal{N}[\![\mathtt{a_1 + a_2}]\!]_\Gamma \ \equiv\ \mathcal{E}[\![\mathtt{a_1}]\!]_\Gamma \ +\ \mathcal{E}[\![\mathtt{a_2}]\!]_\Gamma$$

$$\mathcal{N}[\![\mathtt{a_1 - a_2}]\!]_\Gamma \ \equiv\ \mathcal{E}[\![\mathtt{a_1}]\!]_\Gamma \ -\ \mathcal{E}[\![\mathtt{a_2}]\!]_\Gamma$$

$$\mathcal{N}[\![\mathtt{a_1 * a_2}]\!]_\Gamma \ \equiv\ \mathcal{E}[\![\mathtt{a_1}]\!]_\Gamma \ \cdot\ \mathcal{E}[\![\mathtt{a_2}]\!]_\Gamma$$

$$\mathcal{N}[\![\mathtt{a_1 / a_2}]\!]_\Gamma \ \equiv\ \lfloor \mathcal{E}[\![\mathtt{a_1}]\!]_\Gamma \ /\ \mathcal{E}[\![\mathtt{a_2}]\!]_\Gamma \rfloor, \text{ integer division.}$$

$$\mathcal{N}[\![\mathtt{a_1 \% a_2}]\!]_\Gamma \ \equiv\ \mathcal{E}[\![\mathtt{a_1}]\!]_\Gamma \ \%\ \mathcal{E}[\![\mathtt{a_2}]\!]_\Gamma, \text{ integer remainder.}$$

$$\mathcal{N}[\![\mathtt{-a}]\!]_\Gamma \ \equiv\ -\mathcal{E}[\![\mathtt{a}]\!]_\Gamma$$

$$\mathcal{N}[\![\mathtt{abs\ a}]\!]_\Gamma \ \equiv\ |\mathcal{E}[\![\mathtt{a}]\!]_\Gamma|, \text{ absolute value of an integer.}$$

$$\mathcal{N}[\![\mathtt{card\ s}]\!]_\Gamma \ \equiv\ |\mathcal{E}[\![\mathtt{s}]\!]_\Gamma|, \text{ cardinality of a set.}$$

The `sum` quantifier below ranges over a set E of environments defined by a member `q` of *QuantExpr*. For each member in E, the numerical expression defined by a member `a` of *NumExpr* is evaluated and the result is the sum of all those.

$$\mathcal{N}[\![\mathtt{sum\ (q)(a)}]\!]_\Gamma \ \equiv\ \sum_{\gamma \in \mathcal{Q}[\![\mathtt{q}]\!]_\Gamma} (\mathcal{E}[\![\mathtt{a}]\!]_{\Gamma \cup \gamma})$$

Set Expressions

Evaluating a set expression means evaluating either:

- a given set such as the set of integers or the empty set.
- the elements of an enumerated set of values.
- an integer range expression.
- a set comprehension expression.
- a relation domain expression.

We define the function $\mathcal{S}$ with the following signature:

$$\mathcal{S} \ : \mathit{SetExpr} * \mathit{Environment} \longrightarrow 2^{\mathbb{T}} \cup 2^{2^{\mathbb{T}}}$$

Hence, $\mathcal{S}$ takes as arguments an element of *SetExpr* and an environment, and returns a set of tuples or a set of sets of tuples.

$$\mathcal{S}[\![\texttt{int}]\!]_\Gamma \ \equiv \ \mathbb{Z}$$

$$\mathcal{S}[\![\texttt{nat}]\!]_\Gamma \ \equiv \ \mathbb{N}$$

$$\mathcal{S}[\![\texttt{\{\}}]\!]_\Gamma \ \equiv \ \emptyset$$

$$\mathcal{S}[\![\texttt{\{e}_1\texttt{,...,e}_n\texttt{\}}]\!]_\Gamma \ \equiv \ \{\mathcal{E}[\![\texttt{e}_1]\!]_\Gamma, \ldots, \mathcal{E}[\![\texttt{e}_n]\!]_\Gamma\}$$

$$\mathcal{S}[\![\texttt{a}_1\texttt{..a}_2]\!]_\Gamma \ \equiv \ \{x \in \mathbb{Z} \mid \mathcal{E}[\![\texttt{a}_1]\!]_\Gamma \ \leq \ x \ \leq \ \mathcal{E}[\![\texttt{a}_2]\!]_\Gamma\}$$

$$\mathcal{S}[\![\texttt{\{e |}x_{1_1}\texttt{\&}\cdots\texttt{\&}x_{1_k} \texttt{ : s}_1\texttt{/\textbackslash}\cdots\texttt{/\textbackslash}x_{n_1}\texttt{\&}\cdots\texttt{\&}\ x_{n_l} \texttt{ : s}_n\texttt{\}}]\!]_\Gamma \ \equiv$$
$$\{\mathcal{E}[\![\texttt{e}]\!]_{\Gamma\cup\gamma} \mid \gamma \in \mathcal{Q}[\![x_{1_1}\texttt{\&}\cdots\texttt{\&}x_{1_k} \texttt{ : s}_1\texttt{,...,}x_{n_1}\texttt{\&}\cdots\texttt{\&}\ x_{n_l} \texttt{ : s}_n]\!]_\Gamma\}$$

$$\mathcal{S}[\![\texttt{\{e |}x_{1_1}\texttt{\&}\cdots\texttt{\&}x_{1_k} \texttt{ : s}_1\texttt{/\textbackslash}\cdots\texttt{/\textbackslash}x_{n_1}\texttt{\&}\cdots\texttt{\&}\ x_{n_l} \texttt{ : s}_n \texttt{ | b\}}]\!]_\Gamma \ \equiv$$
$$\{\mathcal{E}[\![\texttt{e}]\!]_{\Gamma\cup\gamma} \mid \gamma \in \mathcal{Q}[\![x_{1_1}\texttt{\&}\cdots\texttt{\&}x_{1_k} \texttt{ : s}_1\texttt{,...,}x_{n_1}\texttt{\&}\cdots\texttt{\&}\ x_{n_l} \texttt{ : s}_n \texttt{ | b}]\!]_\Gamma\}$$

$$\mathcal{S}[\![\texttt{s}_1 \texttt{ \# } \texttt{s}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{s}_1]\!]_\Gamma \times \mathcal{E}[\![\texttt{s}_2]\!]_\Gamma$$

$$\mathcal{S}[\![\texttt{s [m]}]\!]_\Gamma \ \equiv \ \{s \ \mid s \subseteq \mathcal{E}[\![\texttt{s}]\!]_\Gamma \ \wedge \ |s| \in \mathcal{E}[\![\texttt{m}]\!]_\Gamma\}, \text{ when } \texttt{m} \in \mathit{SetExpr}.$$

$$\mathcal{S}[\![\texttt{s [m]}]\!]_\Gamma \ \equiv \ \{s \ \mid s \subseteq \mathcal{E}[\![\texttt{s}]\!]_\Gamma \ \wedge \ |s| = \mathcal{E}[\![\texttt{m}]\!]_\Gamma\}, \text{ when } \texttt{m} \in \mathit{NumExpr}.$$

For the rules below, let $\mathrm{m}(R, S_1, M, S_2))$ denote the boolean expression:

$$\forall x \in S_1 : (|\{z \in S_2 \mid R(x, z)\}| \in M)$$

when M denotes a set of values, and the boolean expression:

$$\forall x \in S_1 : (|\{z \in S_2 \mid R(x, z)\}| = M)$$

when M denotes an integer value.

$$\mathcal{S}[\![\mathsf{s}_1\ \texttt{[}\mathsf{m}_1\texttt{\#}\mathsf{m}_2\texttt{]}\ \mathsf{s}_2]\!]_\Gamma \equiv \{R \subseteq \mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma \times \mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma \mid m(R,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma,\mathcal{E}[\![\mathsf{m}_1]\!]_\Gamma,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma) \wedge m(R,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma,\mathcal{E}[\![\mathsf{m}_2]\!]_\Gamma,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma)\}$$

$$\mathcal{S}[\![\mathsf{s}_1\ \texttt{[m\#]}\ \mathsf{s}_2]\!]_\Gamma \equiv \{R \subseteq \mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma \times \mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma \mid m(R,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma,\mathcal{E}[\![\mathsf{m}]\!]_\Gamma,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma) \wedge m(R,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma,\mathbb{N},\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma)\}$$

$$\mathcal{S}[\![\mathsf{s}_1\ \texttt{[\#m]}\ \mathsf{s}_2]\!]_\Gamma \equiv \{R \subseteq \mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma \times \mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma \mid m(R,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma,\mathbb{N},\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma) \wedge m(R,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma,\mathcal{E}[\![\mathsf{m}]\!]_\Gamma,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma)\}$$

$$\mathcal{S}[\![\mathsf{s}_1\ \texttt{[\#]}\ \mathsf{s}_2]\!]_\Gamma \equiv \{R \subseteq \mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma \times \mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma \mid m(R,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma,\mathbb{N},\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma) \wedge m(R,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma,\mathbb{N},\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma)\}$$

$$\mathcal{S}[\![\mathsf{s}_1\ \texttt{[->m]}\ \mathsf{s}_2]\!]_\Gamma \equiv \{R \subseteq \mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma \times \mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma \mid m(R,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma,\{1\},\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma) \wedge m(R,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma,\mathcal{E}[\![\mathsf{m}]\!]_\Gamma,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma)\}$$

$$\mathcal{S}[\![\mathsf{s}_1\ \texttt{[->]}\ \mathsf{s}_2]\!]_\Gamma \equiv \{R \subseteq \mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma \times \mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma \mid m(R,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma,\{1\},\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma) \wedge m(R,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma,\mathbb{N},\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma)\}$$

$$\mathcal{S}[\![\mathsf{s}_1\ \texttt{->}\ \mathsf{s}_2]\!]_\Gamma \equiv \{R \subseteq \mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma \times \mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma \mid m(R,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma,\{1\},\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma) \wedge m(R,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma,\mathbb{N},\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma)\}$$

$$\mathcal{S}[\![\mathsf{s}_1\ \texttt{[+>m]}\ \mathsf{s}_2]\!]_\Gamma \equiv \{R \subseteq \mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma \times \mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma \mid m(R,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma,\{0,1\},\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma) \wedge m(R,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma,\mathcal{E}[\![\mathsf{m}]\!]_\Gamma,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma)\}$$

$$\mathcal{S}[\![\mathsf{s}_1\ \texttt{[+>]}\ \mathsf{s}_2]\!]_\Gamma \equiv \{R \subseteq \mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma \times \mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma \mid m(R,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma,\{0,1\},\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma) \wedge m(R,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma,\mathbb{N},\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma)\}$$

$$\mathcal{S}[\![\mathsf{s}_1\ \texttt{+>}\ \mathsf{s}_2]\!]_\Gamma \equiv \{R \subseteq \mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma \times \mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma \mid m(R,\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma,\{0,1\},\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma) \wedge m(R,\mathcal{E}[\![\mathsf{s}_2]\!]_\Gamma,\mathbb{N},\mathcal{E}[\![\mathsf{s}_1]\!]_\Gamma)\}$$

Formulas

Evaluating a formula means evaluating a boolean constant, or evaluating the arguments of some relation- or boolean operator and to apply the operator to the resulting values. We define the function $\mathcal{F}$ with the following signature:

$$\mathcal{F} \ : \mathit{Formula} * \mathit{Environment} \longrightarrow \mathbb{B}$$

Hence, $\mathcal{F}$ takes as arguments an element of the syntactic domain *Formula* and an environment, and returns either *true* or *false*.

$$\mathcal{F}[\![\texttt{true}]\!]_\Gamma \ \equiv \ \mathit{true}$$

$$\mathcal{F}[\![\texttt{false}]\!]_\Gamma \ \equiv \ \mathit{false}$$

$$\mathcal{F}[\![\texttt{b}_1 \ \texttt{/\textbackslash} \ \texttt{b}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{b}_1]\!]_\Gamma \ \wedge \ \mathcal{E}[\![\texttt{b}_2]\!]_\Gamma$$

$$\mathcal{F}[\![\texttt{b}_1 \ \texttt{\textbackslash/} \ \texttt{b}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{b}_1]\!]_\Gamma \ \vee \ \mathcal{E}[\![\texttt{b}_2]\!]_\Gamma$$

$$\mathcal{F}[\![\texttt{b}_1 \ \texttt{=>} \ \texttt{b}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{b}_1]\!]_\Gamma \ \Rightarrow \ \mathcal{E}[\![\texttt{b}_2]\!]_\Gamma$$

$$\mathcal{F}[\![\texttt{b}_1 \ \texttt{<=} \ \texttt{b}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{b}_1]\!]_\Gamma \ \Leftarrow \ \mathcal{E}[\![\texttt{b}_2]\!]_\Gamma$$

$$\mathcal{F}[\![\texttt{b}_1 \ \texttt{<=>} \ \texttt{b}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{b}_1]\!]_\Gamma \ \Leftrightarrow \ \mathcal{E}[\![\texttt{b}_2]\!]_\Gamma$$

$$\mathcal{F}[\![\texttt{a}_1 \ \texttt{<} \ \texttt{a}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{a}_1]\!]_\Gamma < \mathcal{E}[\![\texttt{a}_2]\!]_\Gamma$$

$$\mathcal{F}[\![\texttt{a}_1 \ \texttt{=<} \ \texttt{a}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{a}_1]\!]_\Gamma \leq \mathcal{E}[\![\texttt{a}_2]\!]_\Gamma$$

$$\mathcal{F}[\![\texttt{a}_1 \ \texttt{=} \ \texttt{a}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{a}_1]\!]_\Gamma = \mathcal{E}[\![\texttt{a}_2]\!]_\Gamma$$

$$\mathcal{F}[\![\texttt{a}_1 \ \texttt{>=} \ \texttt{a}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{a}_1]\!]_\Gamma \geq \mathcal{E}[\![\texttt{a}_2]\!]_\Gamma$$

$$\mathcal{F}[\![\texttt{a}_1 \ \texttt{>} \ \texttt{a}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{a}_1]\!]_\Gamma > \mathcal{E}[\![\texttt{a}_2]\!]_\Gamma$$

$$\mathcal{F}[\![\texttt{a}_1 \ \texttt{!=} \ \texttt{a}_2]\!]_\Gamma \ \equiv \ \mathcal{E}[\![\texttt{a}_1]\!]_\Gamma \neq \mathcal{E}[\![\texttt{a}_2]\!]_\Gamma$$

The `forall` quantifier ranges over a set E of environments defined by a member `q` of *QuantExpr*. Evaluating the formula `b` with a member of E as an extension of a given Γ must evaluate to *true* for all members of E in order for the whole expression to evaluate to *true*.

$$\mathcal{F}[\![\texttt{forall (q)(b)}]\!]_\Gamma \ \equiv \ \forall \gamma \in \mathcal{Q}[\![\texttt{q}]\!]_\Gamma(\mathcal{E}[\![\texttt{b}]\!]_{\Gamma \cup \gamma})$$

The `exists` quantifier ranges over a set E of environments defined by a member `q` of *QuantExpr*. In order for the expression to evaluate to *true*, the size of E must be larger than 0.

$$\mathcal{F}[\![\texttt{exists (q)}]\!]_\Gamma \;\equiv\; |\mathcal{Q}[\![\texttt{q}]\!]_\Gamma| > 0$$

The `count` quantifier is a generalisation of the `exists` quantifier. Hence, it ranges over a set E of environments defined by a member `q` of *QuantExpr*. In order for the expression to evaluate to *true*, the size of E must be in the set of integers defined by a member `s` of *SetExpr*.

$$\mathcal{F}[\![\texttt{count (s)(q)}]\!]_\Gamma \;\equiv\; |\mathcal{Q}[\![\texttt{q}]\!]_\Gamma| \in \mathcal{E}[\![\texttt{s}]\!]_\Gamma$$

Quantification

Evaluating a quantified expression means generating a set of environments that is used in, e.g., a `forall` quantification. We define the function $\mathcal{Q}$ with the following signature:

$$\mathcal{Q} \;:\; \mathit{QuantExpr} * \mathit{Environment} \longrightarrow 2^{2^{\mathit{Environment}}}$$

Hence, $\mathcal{Q}$ takes as arguments a member of *QuantExpr* and an environment, and returns a set of environments.

$$\begin{aligned}
&\mathcal{Q}[\![x_{1_1}\texttt{\&}\cdots\texttt{\&}x_{1_k} \texttt{ : } \texttt{s}_1\texttt{,}\ldots\texttt{,}x_{n_1}\texttt{\&}\cdots\texttt{\&}\; x_{n_l} \texttt{ : } \texttt{s}_n]\!]_\Gamma \;\equiv\\
&\qquad \{\{x_{1_1} \mapsto e_{1_1}, \ldots, x_{1_k} \mapsto e_{1_k}, \ldots, x_{n_1} \mapsto e_{n_1}, \ldots, x_{n_l} \mapsto e_{n_l}\} \mid\\
&\qquad\; (e_{1_1}, \ldots, e_{1_k}) \in (\mathcal{E}[\![\texttt{s}_1]\!]_\Gamma)^k \wedge \cdots \wedge (e_{n_1}, \ldots, e_{n_l}) \in (\mathcal{E}[\![\texttt{s}_n]\!]_\Gamma)^l\}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{Q}[\![x_{1_1}\texttt{\&}\cdots\texttt{\&}x_{1_k} \texttt{ : } \texttt{s}_1\texttt{,}\ldots\texttt{,}x_{n_1}\texttt{\&}\cdots\texttt{\&}\; x_{n_l} \texttt{ : } \texttt{s}_n \texttt{ | } \texttt{b}]\!]_\Gamma \;\equiv\\
&\qquad \{\{x_{1_1} \mapsto e_{1_1}, \ldots, x_{1_k} \mapsto e_{1_k}, \ldots, x_{n_1} \mapsto e_{n_1}, \ldots, x_{n_l} \mapsto e_{n_l}\} \mid\\
&\qquad\; (e_{1_1}, \ldots, e_{1_k}) \in (\mathcal{E}[\![\texttt{s}_1]\!]_\Gamma)^k \wedge \cdots \wedge (e_{n_1}, \ldots, e_{n_l}) \in (\mathcal{E}[\![\texttt{s}_n]\!]_\Gamma)^l \wedge\\
&\qquad\; \mathcal{E}[\![\texttt{b}]\!]_{\Gamma \cup \{x_{1_1} \mapsto e_{1_1}, \ldots, x_{1_k} \mapsto e_{1_k}, \ldots, x_{n_1} \mapsto e_{n_1}, \ldots, x_{n_l} \mapsto e_{n_l}\}}\}
\end{aligned}$$

For the rules below, let $\diamond \in \{\texttt{<}, \texttt{=<}, \texttt{>}, \texttt{>=}, \texttt{=}, \texttt{!=}\}$, and let $\mathcal{R}$ be the function defined by the set of mappings: $\{(\texttt{<} \mapsto <), (\texttt{=<} \mapsto \leq), (\texttt{>} \mapsto >), (\texttt{>=} \mapsto \geq), (\texttt{=} \mapsto =), (\texttt{!=} \mapsto \neq)\}$.

$$\begin{aligned}
&\mathcal{Q}[\![x \;\diamond\; y \texttt{ : } \texttt{s}]\!]_\Gamma \;\equiv\\
&\qquad \{\{x \mapsto e_1, y \mapsto e_2\} \;\mid (e_1, e_2) \in (\mathcal{E}[\![\texttt{s}]\!]_\Gamma)^2 \;\wedge\; e_1 \;\mathcal{R}(\diamond)\; e_2 \;\}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{Q}[\![x \;\diamond\; y \texttt{ : } \texttt{s} \texttt{ | } \texttt{b}]\!]_\Gamma \;\equiv\\
&\qquad \{\{x \mapsto e_1, y \mapsto e_2\} \;\mid (e_1, e_2) \in (\mathcal{E}[\![\texttt{s}]\!]_\Gamma)^2 \;\wedge\; e_1 \;\mathcal{R}(\diamond)\; e_2 \;\wedge\\
&\qquad\qquad\qquad \mathcal{E}[\![\texttt{b}]\!]_{\Gamma \cup \{x \mapsto e_1, y \mapsto e_2\}}\}
\end{aligned}$$

$$\mathcal{Q}[\![x \;\diamond\; \texttt{e} \texttt{ : } \texttt{s}]\!]_\Gamma \;\equiv\; \{\{x \mapsto e\} \;\mid e \in \mathcal{E}[\![\texttt{s}]\!]_\Gamma \;\wedge\; e \;\mathcal{R}(\diamond)\; \mathcal{E}[\![\texttt{e}]\!]_\Gamma\}$$

$$\begin{aligned}
&\mathcal{Q}[\![x \;\diamond\; \texttt{e} \texttt{ : } \texttt{s} \texttt{ | } \texttt{b}]\!]_\Gamma \;\equiv\\
&\qquad \{\{x \mapsto e\} \;\mid e \in \mathcal{E}[\![\texttt{s}]\!]_\Gamma \;\wedge\; e \;\mathcal{R}(\diamond)\; \mathcal{E}[\![\texttt{e}]\!]_\Gamma \;\wedge\; \mathcal{E}[\![\texttt{b}]\!]_{\Gamma \cup \{x \mapsto e\}}\}
\end{aligned}$$

$$\mathcal{Q}[\![\texttt{e} \;\diamond\; y \texttt{ : } \texttt{s}]\!]_\Gamma \;\equiv\; \{\{y \mapsto e\} \;\mid e \in \mathcal{E}[\![\texttt{s}]\!]_\Gamma \;\wedge\; \mathcal{E}[\![\texttt{e}]\!]_\Gamma \;\mathcal{R}(\diamond)\; e \;\}$$

$$\mathcal{Q}[\![\mathsf{e} \diamond y : \mathsf{s} \mid \mathsf{b}]\!]_\Gamma \equiv \{\{y \mapsto e\} \mid e \in \mathcal{E}[\![\mathsf{s}]\!]_\Gamma \wedge \mathcal{E}[\![\mathsf{e}]\!]_\Gamma \, \mathcal{R}(\diamond)\, e \wedge \mathcal{E}[\![\mathsf{b}]\!]_{\Gamma \cup \{y \mapsto e\}}\}$$

$$\mathcal{Q}[\![\mathsf{e}_1 \diamond \mathsf{e}_2 : \mathsf{s}]\!]_\Gamma \equiv \emptyset$$

$$\mathcal{Q}[\![\mathsf{e}_1 \diamond \mathsf{e}_2 : \mathsf{s} \mid \mathsf{b}]\!]_\Gamma \equiv \emptyset$$

Paper A

Introducing ESRA: a Relational Language for Modelling Combinatorial Problems

Reprinted from:

M. Bruynooghe (editor), Revised selected papers of the 13th International Symposium on Logic Based Program Synthesis and Transformation - LOPSTR 2003, pp. 214–232, Lecture Notes in Computer Science, volume 3018. Springer-Verlag, 2004.

Introducing ESRA, a Relational Language for Modelling Combinatorial Problems*

Pierre Flener, Justin Pearson, and Magnus Ågren**

Department of Information Technology
Uppsala University, Box 337, S – 751 05 Uppsala, Sweden
{pierref,justin,agren}@it.uu.se

Abstract. Current-generation constraint programming languages are considered by many, especially in industry, to be too low-level, difficult, and large. We argue that solver-independent, high-level relational constraint modelling leads to a simpler and smaller language, to more concise, intuitive, and analysable models, as well as to more efficient and effective model formulation, maintenance, reformulation, and verification. All this can be achieved without sacrificing the possibility of efficient solving, so that even time-pressed or less competent modellers can be well assisted. Towards this, we propose the ESRA relational constraint modelling language, showcase its elegance on some well-known problems, and outline a compilation philosophy for such languages.

1 Introduction

Current-generation constraint programming languages are considered by many, especially in industry, to be too low-level, difficult, and large. Consequently, their solvers are not in as widespread use as they ought to be, and constraint programming is still fairly unknown in many application domains, such as molecular biology. In order to unleash the proven powers of constraint technology and make it available to a wider range of problem modellers, a solver-independent, higher-level, simpler, and smaller modelling notation is needed.

In our opinion, even recent commercial languages such as OPL [31] do not go far enough in that direction. Many common modelling patterns have not been captured in special constructs. They have to be painstakingly spelled out each time, at a high risk for errors, often using low-level devices such as reification.

In recent years, modelling languages based on some logic with sets and relations have gained popularity in formal methods, witness the B [1] and Z [29] specification languages, the ALLOY [16] object modelling language, and the *Object Constraint Language* (OCL) [35] of the *Unified Modelling Language* (UML) [27]. In semantic data modelling this had been long advocated; most notably via entity-relationship-attribute (ERA) diagrams.

* A previous version of this paper appears pages 63–77 in the informally published proceedings of the *Second International Workshop Modelling and Reformulating CSPs*, available at http://www-users.cs.york.ac.uk/~frisch/Reformulation/03/

** The authors' names are ordered according to the Swedish alphabet.

M. Bruynooghe (Ed.): LOPSTR 2004, LNCS 3018, pp. 214–232, 2004.

Sets and set expressions started appearing as modelling devices in some constraint languages. Set variables are often implemented by the set interval representation [13]. In the absence of such an explicit set concept, modellers usually painstakingly represent a set variable by its characteristic function, namely as a sequence of 0/1 integer variables, as long as the size of the domain of the set.

Relations have not received much attention yet in constraint programming languages, except total functions, via arrays. Indeed, a total function f can be represented in many ways [15], say as a 1-dimensional array of variables over the range of f, indexed by its domain, or as a 2-dimensional array of Boolean variables, indexed by the domain and range of f, or as a 1-dimensional array of set variables over the domain of f, indexed by its range, or even with some redundancy. Other than retrieving the (unique) image under a total function of a domain element, there has been no support for relational expressions.

Matrix modelling [8, 10, 31] has been advocated as one way of capturing common modelling patterns. Alternatively, it has been argued [11, 15] that functions, and hence relations, should be supported by an abstract datatype (ADT). It is then *the compiler* that must (help the modeller) choose a suitable representation, say in a contemporary constraint programming language, for each instance of the ADT, using empirically or theoretically gained modelling insights.

We here demonstrate, as originally conjectured in [9], that a suitable first-order relational calculus is a good basis for a high-level, ADT-based, and solver-independent constraint modelling language. It gives rise to very natural and easy-to-maintain models of combinatorial problems. Even in the (temporary) absence of a corresponding high-level search language, this generality does not necessarily come at a loss in solving efficiency, as abstract relational models are devoid of representation details so that the results of analysis can be exploited.

Our aims here are only to justify and present our new language, called ESRA, to illustrate its elegance and the flexibility of its models by some examples, and to argue that it can be compiled into efficient models in lower-level (constraint programming) languages. The syntax, denotational semantics, and type system of the proposed language are discussed in full detail in an online appendix [12] and a second prototype of the advocated compiler is under development.

The rest of this paper is organised as follows. In Section 2, we present our relational language for modelling combinatorial problems and deploy it on three real-life problems before discussing its compilation. This allows us to list, in Section 3, the benefits of relational modelling. Finally, in Section 4, we conclude as well as discuss related and future work.

2 Relational Constraint Modelling with ESRA

In Section 2.1, we justify the design decisions behind our new ESRA constraint modelling language, targeted at constraint programmers. Then, in Section 2.2, we introduce its concepts, syntax, type system, and semantics. Next, in Section 2.3, we deploy ESRA on three real-life problems. Finally, in Section 2.4, we discuss the design of our prototype compilers for ESRA.

2.1 Design Decisions

The key design decisions for our new relational constraint modelling language — called ESRA for *Executable Symbolism for Relational Algebra* — were as follows.

We want to capture common modelling idioms in a new abstract datatype for relations, so as to design a high-level and simple language. The constructs of the language are orthogonal, so as to keep the language small. Computational completeness is not aimed at, as long as the language is useful for elegantly modelling a large number of combinatorial problems.

We focus on *finite*, discrete domains. Relations are built from such domains and sets are viewed as unary relations. Theoretical difficulties are sidestepped by supporting only bounded quantification, but not negation nor sets of sets.

The language has an ASCII syntax, mimicking mathematical and logical notation as closely as possible, as well as a LaTeX-based syntax, especially used for pretty-printing models in that notation.

2.2 Concepts, Syntax, Type System, and Semantics of ESRA

For reasons of space, we only give an informal semantics. The interested reader is invited to consult [12] for a complete description of the language. Essentially, the semantics of the language is a conservative extension of existential second-order logic. Existential quantification of relations is used to assert that relations are to be found that satisfy sets of first-order constraints. This is in contrast with extensions of logic programming [6, 25] where second-order relations can be specified recursively using Horn clauses, which needs a much more careful treatment of the fixed-point semantics.

Code excerpts are here provided out of the semantic context of any particular problem statement, just to illustrate the syntax, but a suggested reading in plain English is always provided. In Section 2.3, we will actually start from plain English problem statements and show how they can be modelled in ESRA. Code excerpts are always given in the pretty-printed form, but we indicate the ASCII notation for every symbol where it necessarily differs.

An ESRA model starts with a sequence of declarations of named *domains* (or types) as well as named *constants* and *decision variables* that are tied to domains. Then comes the *objective*, which is to find values for the decision variables within their domains so that some *constraints* are satisfied and possibly some *cost expression* takes an optimal value.

The Type System. A *primitive domain* is a finite, extensionally given set of new names or integers, comma-separated and enclosed as usual in curly braces. An integer domain can also be given intensionally as a finite integer interval, by separating its lower and upper bounds with '...' (denoted in ASCII by '..'), without using curly braces. When these bounds coincide, the corresponding singleton domain $n \ldots n$ or $\{n\}$ can be abbreviated to n. Context always determines whether an integer n designates itself or the singleton domain $\{n\}$. A domain can also be given intensionally using set comprehension notation.

The only *predefined* primitive domains are the sets $\mathbb{N}$ (denoted in ASCII by '`nat`') and $\mathbb{Z}$ (denoted in ASCII by '`int`'), which are '$0 \ldots \sup$' and '$\inf \ldots \sup$' respectively, where the predefined constant identifiers 'inf' and 'sup' stand for the smallest negative and largest positive representable integers respectively. *User-defined* primitive domains are declared after the 'dom' keyword and initialised at compile-time, using the '=' symbol, or at run-time, via a datafile, otherwise interactively.

Example 1. The statement

$$\text{dom } \mathit{Varieties}, \mathit{Blocks}$$

declares two domains called *Varieties* and *Blocks* that are to be initialised at run-time. As in OPL [31], this neatly separates the problem model from its instance data, so that the actual constraint satisfaction problem is obtained at run-time.

Similarly, the statement

$$\text{dom } \mathit{Players} = 1 \ldots g * s,\ \ \mathit{Weeks} = 1 \ldots w,\ \ \mathit{Groups} = 1 \ldots g$$

where g, s, w are integer-constant identifiers (assumed previously declared, in a way shown below), declares integer domains called *Players*, *Weeks*, and *Groups* that are initialised at compile-time.

Finally, the declaration

$$\text{dom } \mathit{Even} = \{i \mid i : 0 \ldots 100 \mid i \ \% \ 2 = 0\}$$

initialises the domain *Even* of all even natural numbers up to 100.

The usual binary infix $\times$ constructor (denoted in ASCII by '`#`') allows the construction of Cartesian products.

The only *constructed domains* are relational domains. In order to simultaneously capture frequently occurring multiplicity constraints on relations, we offer a parameterised binary infix $\times$ domain constructor. The relational domain $A \ {}^{M_1}\!\times^{M_2} B$, where A and B are (possibly Cartesian products of) primitive domains, designates a set of binary relations in $A \times B$. The optional M_1 and M_2, called *multiplicities*, must be integer sets and have the following semantics: for every element a of A, the number of elements of B related to a must be in M_1, while for every element b of B, the number of elements of A related to b must be in M_2 [1]. An omitted multiplicity stands for $\mathbb{N}$.

Example 2. The constructed domain

$$\mathit{Varieties} \ {}^{r}\!\times^{k} \mathit{Blocks}$$

designates the set of all relations in $\mathit{Varieties} \times \mathit{Blocks}$ where every variety occurs in exactly r blocks and every block contains exactly k varieties. These are two occurrences where an integer abbreviates the singleton domain containing it.

[1] Note that our syntax is the opposite of the UML one, say, where the multiplicities are written in the other order, with the *same* semantics. That convention can however *not* be usefully upgraded to Cartesian products of arity higher than 2.

In the absence of such facilities for relations and their multiplicities, a relational domain would have to be modelled using arrays, say. This may be a premature commitment to a concrete data structure, as the modeller may not know yet, especially prior to experimentation, which particular (array-based) representation of a relational decision variable will lead to the most efficient solving. The problem constraints, including the multiplicities, would have to be formulated in the constraints part of the model, based on the chosen representation. If the experiments revealed that another representation should be tried, then the modeller would have to first painstakingly reformulate the declaration of the decision variable as well as all its constraints. Our ADT view of relations overcomes this flaw: it is now *the compiler* that must (help the modeller) choose a suitable representation for each instance of the ADT by using empirically or theoretically gained insights. Also, multiplicities need not become counting constraints, but are succinctly and conveniently captured in the declaration.

We view sets as unary relations: $A\ M$, where A is a domain and M an integer set, constructs the domain of all subsets of A whose cardinality is in M. The multiplicity M is mandatory here; otherwise there would be ambiguity whether a value of the domain A is an element or an arbitrarily sized subset of A.

For total and partial functions, the left-hand multiplicity M_1 is $1 \dots 1$ and $0 \dots 1$ respectively. In order to dispense with these left-hand multiplicities for total and partial functions, we offer the usual $\longrightarrow$ and $\not\longrightarrow$ (denoted in ASCII by '`->`' and '`+>`') domain constructors respectively, as shorthands. They may still have right-hand multiplicities though.

For injections, surjections, and bijections, the right-hand multiplicity M_2 is $0 \dots 1$, $1 \dots \mathrm{sup}$, and $1 \dots 1$ respectively. Rather than elevating these particular cases of functions to first-class concepts with an invented specific syntax in ESRA, we prefer keeping our language lean and close to mathematical notation.

Example 3. The constructed domain

$$(\mathit{Players} \times \mathit{Weeks}) \longrightarrow^{s*w} \mathit{Groups}$$

designates the set of all total functions from $\mathit{Players} \times \mathit{Weeks}$ into *Groups* such that every group is related to exactly sw (player,week) pairs.

We provide no support (yet) for bags and sequences, as relations provide enough challenges for the time being. Note that a *bag* can be modelled as a total function from its domain into $\mathbb{N}$, giving the repetition count of each element. Similarly, a *sequence* of length n can be modelled as a total function from $1 \dots n$ into its domain, telling which element is at each position. This does *not* mean that the representation of bags and sequences is fixed (to the one of total functions), because, as we shall see in Section 2.4, the various relations (and thus total functions) of a model need not have the same representation.

Modelling the Instance Data and Decision Variables. All identifier declarations are strongly typed and denote variables that are implicitly universally

quantified over the entire model, with the constants expected to be ground before search begins while the decision variables can still be unbound at that moment.

Like the user-defined primitive domains, constants help describe the instance data of a problem. A constant identifier is declared after the 'cst' keyword and is tied to its domain by ':', meaning set membership. Constants are initialised at compile-time, using the '=' symbol, or at run-time, via a datafile, otherwise interactively. Again, run-time initialisation provides a neat separation of problem models and problem instances.

Example 4. The statement

$$\text{cst } r, k, \lambda : \mathbb{N}$$

declares three natural number constants that are to be initialised at run-time.

As already seen in Examples 2 and 3, the availability of total functions makes arrays unnecessary. The statement

$$\text{cst } \mathit{CrewSize} : \mathit{Guests} \longrightarrow \mathbb{N},\ \mathit{SpareCap} : \mathit{Hosts} \longrightarrow \mathbb{N}$$

declares two natural-number functions, to be provided at run-time.

A decision-variable identifier is declared after the 'var' keyword and is tied to its domain by ':'.

Example 5. The statement

$$\text{var } \mathit{BIBD} : \mathit{Varieties}\ {}^{r}\!\times^{k}\ \mathit{Blocks}$$

declares a relation called *BIBD* of the domain of Example 2.

Modelling the Cost Expression and the Constraints. *Expressions* and first-order logic *formulas* are constructed in the usual way.

For *numeric expressions*, the arguments are either integers or identifiers of the domain $\mathbb{N}$ or $\mathbb{Z}$, including the predefined constants 'inf' and 'sup'. Usual unary ($-$, 'abs' for absolute value, and 'card' for the cardinality of a set expression), binary infix ($+$, $-$, $*$, / for integer quotient, and % for integer remainder), and aggregate ($\sum$, denoted in ASCII by '`sum`') arithmetic operators are available. A sum is indexed by local variables ranging over finite sets, which may be filtered on-the-fly by a condition given after the '|' symbol (read 'such that').

Sets obey the same rules as domains. So, for *set expressions*, the arguments are either set identifiers or (intensionally or extensionally) given sets, including the predefined sets $\mathbb{N}$ and $\mathbb{Z}$. Only the (unparameterised) binary infix domain constructor $\times$ and its specialisations $\longrightarrow$ and $\not\longrightarrow$ are available as operators.

Finally *function expressions* are built by applying a function identifier to an argument tuple. We have found no use yet for any other operators on functions (but see the discussion of future work in Section 4).

Example 6. The numeric expression

$$\sum_{g:\mathit{Guests}\ \mid\ \mathit{Schedule}(g,p)=h} \mathit{CrewSize}(g)$$

denotes the sum of the crew sizes of all the guest boats that are scheduled to visit host h at period p, assuming this expression is within the scope of the local variables h and p. The nested function expression $\mathit{CrewSize}(g)$ stands for the size of the crew of guest g, which is a natural number according to Example 4.

Atoms are built from numeric expressions with the usual comparison predicates, such as the binary infix $=$, $\neq$, and $\leq$ (denoted in ASCII by '`=`', '`!=`', and '`=<`' respectively). Atoms also include the predefined 'true' and 'false', as well as references to the elements of a relation. We have found no use yet for any other predicates. Note that '$\in$' is unnecessary as $x \in S$ is equivalent to $S(x)$.

Example 7. The atom $\mathit{BIBD}(v_1, i)$ stands for the truth value of variety v_1 being related to block i in the *BIBD* relation of Example 5.

Formulas are built from atoms. The usual binary infix connectives ($\wedge$, $\vee$, $\Rightarrow$, $\Leftarrow$, and $\Leftrightarrow$, denoted in ASCII by '`/\`', '`\/`', '`=>`', '`<=`', and '`<=>`' respectively) and quantifiers ($\forall$ and $\exists$, denoted in ASCII by '`forall`' and '`exists`' respectively) are available. A quantified formula is indexed by local variables ranging over finite sets, which may be filtered on-the-fly by a condition given after the '|' symbol (read 'such that'). As we provide a rich (enough) set of predicates, we are only interested in models that can be formulated positively, and thus dispense with the negation connective. The usual typing and precedence rules for operators and connectives apply. All binary operators associate to the left.

Example 8. The formula

$$\forall(p : \mathit{Periods},\ h : \mathit{Hosts}) \left(\sum_{g:\mathit{Guests} \ \mid \ \mathit{Schedule}(g,p)=h} \mathit{CrewSize}(g) \right) \leq \mathit{SpareCap}(h)$$

constrains the spare capacity of any host boat h not to be exceeded at any period p by the sum of the crew sizes of all the guest boats that are scheduled to visit host h at period p.

A generalisation of the $\exists$ quantifier turns out to be very useful. We define

$$\mathrm{count}(\mathit{Multiplicity})(x : \mathit{Set} \mid \mathit{Condition})$$

to hold if and only if the cardinality of the set comprehension $\{x : \mathit{Set} | \mathit{Condition}\}$ is in the integer set *Multiplicity*. So

$$\exists(x : \mathit{Set} \mid \mathit{Condition})$$

is actually syntactic sugar for

$$\mathrm{count}(1 \ldots \mathrm{sup})(x : \mathit{Set} \mid \mathit{Condition})$$

Example 9. The formula

$$\forall(v_1 < v_2 : \mathit{Varieties})\ \mathrm{count}(\lambda)(j : \mathit{Blocks} \mid \mathit{BIBD}(v_1, j) \wedge \mathit{BIBD}(v_2, j))$$

says that each ordered pair of varieties v_1 and v_2 occurs together in exactly λ blocks, via the *BIBD* relation. Regarding the excerpt '$v_1 < v_2 : \mathit{Varieties}$', note that multiple local variables can be quantified at the same time, and that a filtering condition on them may then be pushed across the '|' symbol.

Example 10. Assuming that the function *Schedule* is of the domain of Example 3 and thus returns a group, the formula

$$\forall(p_1 < p_2 : \mathit{Players})\ \mathrm{count}(0 \ldots 1)(v : \mathit{Weeks} \mid \mathit{Schedule}(p_1, v) = \mathit{Schedule}(p_2, v))$$

says that there is at most one week where any ordered pair of players p_1 and p_2 is scheduled to play in the same group.

A *cost expression* is a numeric expression that has to be optimised. The *constraints* on the decision variables of a model are a conjunction of formulas, using $\wedge$ as the connective. The *objective* of a model is either to solve its constraints:

$$\text{solve } \mathit{Constraints}$$

or to minimise the value of its cost expression subject to its constraints:

$$\text{minimise } \mathit{CostExpression} \text{ such that } \mathit{Constraints}$$

or similarly for maximising. A *model* consists of a sequence of domain, constant, and decision-variable declarations followed by an objective, without separators.

Example 11. Putting together code fragments from Examples 1, 4, 5, and 9, we obtain the model of Figure 2 two pages ahead, discussed in Section 2.3.

The grammar of ESRA is described in Figure 1. For brevity and ease of reading, we have omitted most syntactic-sugar options as well as the rules for identifiers, names, and numbers. The notation $\langle nt \rangle^{s^*}$ stands for a sequence of zero or more occurrences of the non-terminal $\langle nt \rangle$, separated by symbol s. Similarly, $\langle nt \rangle^{s^+}$ stands for one or more occurrences of $\langle nt \rangle$, separated by s. The typing rules ensure that the equality predicates $=$ and $\neq$ are only applied to expressions of the same type, that the other comparison predicates, such as $\leq$, are only applied to numeric expressions, and so on.

2.3 Examples

We now showcase the elegance and flexibility of our language on three real-life problems, namely Balanced Incomplete Block Designs, the Social Golfers problem, and the Progressive Party problem.

Balanced Incomplete Block Designs. Let V be any set of v elements, called *varieties*. A *balanced incomplete block design* (BIBD) is a bag of b subsets of V, called *blocks*, each of size k (constraint C_1), such that each pair of distinct

```
⟨Model⟩ ::= ⟨Decl⟩+ ⟨Objective⟩

⟨Decl⟩ ::= ⟨DomDecl⟩ | ⟨CstDecl⟩ | ⟨VarDecl⟩

⟨DomDecl⟩ ::= dom ⟨Id⟩ [ = ⟨Set⟩ ]

⟨CstDecl⟩ ::= cst ⟨Id⟩ [ = ⟨Tuple⟩ | ⟨Set⟩ ] : ⟨SetExpr⟩

⟨VarDecl⟩ ::= var ⟨Id⟩ : ⟨SetExpr⟩

⟨Objective⟩ ::= solve ⟨Formula⟩
              | ( minimise | maximise ) ⟨NumExpr⟩ such that ⟨Formula⟩

⟨Expr⟩ ::= ⟨Id⟩ | ⟨Name⟩ | ⟨Tuple⟩ | ⟨NumExpr⟩ | ⟨SetExpr⟩ | ⟨FuncAppl⟩ | ( ⟨Expr⟩ )

⟨NumExpr⟩ ::= ⟨Id⟩ | ⟨Int⟩ | ⟨Nat⟩ | inf | sup | ⟨FuncAppl⟩
            | ⟨NumExpr⟩ ( + | - | * | / | % ) ⟨NumExpr⟩
            | ( - | abs ) ⟨NumExpr⟩
            | card ⟨SetExpr⟩
            | sum ( ⟨QuantExpr⟩ ) ( ⟨NumExpr⟩ )

⟨SetExpr⟩ ::= ⟨Set⟩ | ⟨SetExpr⟩ [⟨Set⟩]
            | ⟨SetExpr⟩ ( [⟨Set⟩]#[⟨Set⟩] | # ) ⟨SetExpr⟩
            | ⟨SetExpr⟩ ( [->[⟨Set⟩]] | -> | [+>[⟨Set⟩]] | +> ) ⟨SetExpr⟩

⟨Set⟩ ::= ⟨Id⟩ | int | nat
        | { ⟨Tuple⟩,* } | { ⟨ComprExpr⟩ }
        | ⟨NumExpr⟩..⟨NumExpr⟩ | ⟨NumExpr⟩

⟨ComprExpr⟩ ::= ⟨Expr⟩ | ( ⟨IdTuple⟩&+ in ⟨SetExpr⟩ )/\+ [ | ⟨Formula⟩ ]

⟨FuncAppl⟩ ::= ⟨Id⟩ ⟨Tuple⟩

⟨Tuple⟩ ::= (⟨Expr⟩,+) | ⟨Expr⟩

⟨Formula⟩ ::= true | false | ⟨RelAppl⟩
            | ⟨Formula⟩ ( /\ | \/ | => | <= | <=> ) ⟨Formula⟩
            | ⟨NumExpr⟩ ( < | =< | = | >= | > | != ) ⟨NumExpr⟩
            | forall ( ⟨QuantExpr⟩ ) ( ⟨Formula⟩ )
            | count ( ⟨Set⟩ ) ( ⟨QuantExpr⟩ )

⟨RelAppl⟩ ::= ⟨Id⟩ ⟨Tuple⟩

⟨QuantExpr⟩ ::= ( ( ⟨RelQvars⟩ | ⟨IdTuple⟩&+ ) in ⟨SetExpr⟩ ),+ [ | ⟨Formula⟩ ]

⟨RelQvars⟩ ::= ⟨Expr⟩ ( < | =< | = | >= | > | != ) ⟨Expr⟩

⟨IdTuple⟩ ::= ⟨Id⟩ | ( ⟨Id⟩,+ )
```

Fig. 1. The grammar of ESRA

varieties occurs together in exactly λ blocks (C_2), with $2 \leq k < v$. An implied constraint is that each variety occurs in the same number of blocks (C_3), namely $r = \lambda(v-1)/(k-1)$. A BIBD is parameterised by a 5-tuple $\langle v, b, r, k, \lambda \rangle$ of

dom *Varieties*, *Blocks*
cst $r, k, \lambda : \mathbb{N}$
var *BIBD* : *Varieties* $^{r}\times^{k}$ *Blocks*
solve
$\forall(v_1 < v_2 : \textit{Varieties})\ \text{count}(\lambda)(j : \textit{Blocks} \mid \textit{BIBD}(v_1, j) \wedge \textit{BIBD}(v_2, j))$

Fig. 2. A pretty-printed ESRA model for BIBDs

```
dom Varieties, Blocks
cst r, k, lambda : nat
var BIBD : Varieties [r#k] Blocks
solve
   forall (v1 < v2 : Varieties)
      count (lambda) (j : Blocks | BIBD(v1,j) /\ BIBD(v2,j))
```

Fig. 3. An ESRA model for BIBDs

parameters. Originally intended for the design of statistical experiments, BIBDs also have applications in cryptography and other domains. See Problem 28 at `http://www.csplib.org` for more information.

The instance data can be declared as the two domains *Varieties* and *Blocks*, of implicit sizes v and b respectively, as well as the three natural-number constants r, k, and λ, as in Examples 1 and 4. A unique relational decision variable, *BIBD*, can then be declared as in Example 5, thereby immediately taking care of the constraints C_1 and C_3. The remaining constraint C_2 can be modelled as in Example 9. Figure 2 shows the resulting pretty-printed ESRA model, while Figure 3 shows it in ASCII notation.

For comparison, an OPL [31] model is shown in Figure 4, where '`= ...`' means that the value is to be found in a corresponding datafile. The decision variable `BIBD` is a 2-dimensional array of integers `0` or `1`, indexed by the varieties and blocks, such that `BIBD[i,j]` $=$ `1` iff variety `i` is contained in block `j`. Furthermore, the constraints C_1 and C_3, which we could capture by multiplicities in the ESRA model, need here to be stated in more length. Finally, the constraint C_2 is stated using a higher-order constraint[2]: for each ordered pair of varieties `v1` and `v2`, the number of times they appear in the same block, that is the number of blocks `j` where `BIBD(v1,j)` $=$ `1` $=$ `BIBD(v2,j)` holds, must equal `lambda`.

In an OPL model, one needs to decide what concrete datatypes to use for representing the abstract decision variables of the original problem statement. In this case, we chose a 2-dimensional `0/1` array `BIBD`, indexed by `Varieties` and `Blocks`. We could just as well have chosen a different representation, say (if OPL had set variables) a 1-dimensional array `BIBD`, indexed by `Blocks`, of subsets of `Varieties`. Such a choice affects the formulation of every constraint and the cost expression, but is premature as even expert intuition is weak in predicting which representation choice leads to the best solving efficiency. Consequently, the modeller has to frequently reformulate the constraints and the cost expression

[2] A higher-order constraint refers to the truth value of another constraint. In OPL, the latter is nested in parentheses, truth is represented by 1, and falsity by 0.

```
enum Varieties = ..., Blocks = ...;
int r = ...; int k = ...; int lambda = ...;
range Boolean 0..1;
var Boolean BIBD[Varieties,Blocks];
solve {
   forall(j in Blocks) sum(i in Varieties) BIBD[i,j] = k;
   forall(i in Varieties) sum(j in Blocks) BIBD[i,j] = r;
   forall(ordered v1,v2 in Varieties)
      sum(j in Blocks) (BIBD[v1,j] = 1 = BIBD[v2,j]) = lambda;
   ... symmetry-breaking code ...
};
```

Fig. 4. An OPL model for BIBDs

while experimenting with different representations. No such choices have to be made in an ESRA model, making ESRA a more convenient modelling language.

As a consequence to such representation choices, one often introduces an astronomical amount of symmetries into an OPL model that are not present in the original problem statement [10]. For example, given a solution, any two rows or columns in the array `BIBD` can be swapped, giving a different, but symmetrically equivalent, solution. Such symmetries need to be addressed in order to achieve efficient solving. Hence, symmetry-breaking code [10, 32] would have to be inserted, as indicated in Figure 4. Since such choices are postponed to the compilation phase in ESRA (see Section 2.4), any symmetries consciously introduced can be handled (automatically) in that process.

The Social Golfers Problem. In a golf club, there are n players, each of whom plays golf once a week (constraint C_1) and always in g groups of size s (C_2), hence $n = gs$. The objective is to determine whether there is a schedule of w weeks of play for these golfers, such that there is at most one week where any two distinct players are scheduled to play in the same group (C_3). An implied constraint is that every group occurs exactly sw times across the schedule (C_4). See Problem 10 at `http://www.csplib.org` for more information.

The instance data can be declared as the three natural-number constants g, s, and w, via 'cst $g, s, w : \mathbb{N}$', as well as the three domains *Players*, *Weeks*, and *Blocks*, as in Example 1. A unique decision variable, *Schedule*, can then be declared using the functional domain in Example 3, thereby immediately taking care of the constraints C_1 (because of the totality of the function) and C_4. The constraint C_3 can be modelled as in Example 10. The constraint C_2 can be stated using the count quantifier, as seen in the pretty-printed ESRA model of Figure 5.

Note the different style of modelling sets of unnamed objects, via the separation of models from the instance data, compared to Figure 2. There we introduce two sets without initialising them at the model level, while here we introduce three uninitialised constants that are then used to arbitrarily initialise three domains of desired cardinalities. Both models can be reformulated in the other style. The benefit of such sets of unnamed objects is that their elements are indistinguishable, so that lower-level representations of relational decision variables whose domains involve such sets are known to introduce symmetries.

$$
\begin{array}{l}
\text{cst } g, s, w : \mathbb{N} \\
\text{dom } \mathit{Players} = 1 \ldots g * s,\ \mathit{Weeks} = 1 \ldots w,\ \mathit{Groups} = 1 \ldots g \\
\text{var } \mathit{Schedule} : (\mathit{Players} \times \mathit{Weeks}) \longrightarrow^{s*w} \mathit{Groups} \\
\text{solve} \\
\quad \forall (p_1 < p_2 : \mathit{Players})\ \text{count}(0 \ldots 1)(v : \mathit{Weeks} \mid \mathit{Schedule}(p_1, v) = \mathit{Schedule}(p_2, v)) \\
\quad \wedge\ \forall (h : \mathit{Groups}, v : \mathit{Weeks})\ \text{count}(s)(p : \mathit{Players} \mid \mathit{Schedule}(p, v) = h)
\end{array}
$$

Fig. 5. A pretty-printed ESRA model for the Social Golfers problem

$$
\begin{array}{l}
\text{dom } \mathit{Guests},\ \mathit{Hosts},\ \mathit{Periods} \\
\text{cst } \mathit{SpareCap} : \mathit{Hosts} \longrightarrow \mathbb{N},\ \mathit{CrewSize} : \mathit{Guests} \longrightarrow \mathbb{N} \\
\text{var } \mathit{Schedule} : (\mathit{Guests} \times \mathit{Periods}) \longrightarrow \mathit{Hosts} \\
\text{solve} \\
\quad \forall (p : \mathit{Periods},\ h : \mathit{Hosts}) \left(\sum_{g : \mathit{Guests} \ \mid \ \mathit{Schedule}(g,p) = h} \mathit{CrewSize}(g) \right) \leq \mathit{SpareCap}(h) \\
\quad \wedge\ \forall (g : \mathit{Guests}, h : \mathit{Hosts})\ \text{count}(0 \ldots 1)(p : \mathit{Periods} \mid \mathit{Schedule}(g, p) = h) \\
\quad \wedge\ \forall (g_1 < g_2 : \mathit{Guests})\ \text{count}(0 \ldots 1)(p : \mathit{Periods} \mid \mathit{Schedule}(g_1, p) = \mathit{Schedule}(g_2, p))
\end{array}
$$

Fig. 6. A pretty-printed ESRA model for the Progressive Party problem

The Progressive Party Problem. The problem is to timetable a party at a yacht club. Certain boats are designated as hosts, while the crews of the remaining boats are designated as guests. The crew of a host boat remains on board throughout the party to act as hosts, while the crew of a guest boat together visits host boats over a number of periods. The spare capacity of any host boat is not to be exceeded at any period by the sum of the crew sizes of all the guest boats that are scheduled to visit it then (constraint C_1). Any guest crew can visit any host boat in at most one period (C_2). Any two distinct guest crews can visit the same host boat in at most one period (C_3). See Problem 13 at `http://www.csplib.org` for more information.

The instance data can be declared as the three domains *Guests*, *Hosts*, and *Periods*, via 'dom *Guests*, *Hosts*, *Periods*', as well as the two functional constants *SpareCap* and *CrewSize*, as in Example 4. A unique functional decision variable, *Schedule*, can then be declared via 'var $\mathit{Schedule} : (\mathit{Guests} \times \mathit{Periods}) \longrightarrow \mathit{Hosts}$'. The constraint C_1 can now be modelled as in Example 8. The constraints C_2 and C_3 can be stated using the count quantifier, as seen in the pretty-printed ESRA model of Figure 6.

2.4 Compiling Relational Models

A compiler for ESRA is currently under development. It is being written in OCAML (`http://www.ocaml.org`) and compiles ESRA models into SICStus Prolog [5] finite-domain constraint programs. Our choice of target language is motivated by its excellent collection of global constraints and by our collaboration with its developers on designing new global constraints.

We already have an ESRA-to-OPL compiler [36, 15], written in Java, for a restriction of ESRA to functions, now called Functional-ESRA. That project gave us much of the expertise needed for developing the current compiler.

The solver-independent ESRA language is so high-level that it is very small compared to such target languages, especially in the number of necessary primitive constraints. The full panoply of features of such target languages can, and must, be deployed during compilation. In particular, the implementation of decision-variable indices into matrices is well-understood.

In order to bootstrap our new compiler quickly, we decided to represent initially *every* relational decision variable by a matrix of 0/1 variables, indexed by its participating sets. This first version of the new compiler is thus deterministic.

The plan is then to add alternatives to this unique representation rule, depending on the multiplicities and other constraints on the relation, achieving a *non-deterministic compiler*, such as our existing Functional-ESRA-to-OPL compiler [36, 15]. The modeller is then invited to experiment with her (real-life) instance data and the resulting compiled programs, so as to determine which one is the 'best'. If the compiler is provided with those instance data, then it can be extended to automate such experiments and generate rankings.

Eventually, more intelligence will be built into the compiler via *heuristics* (such as those of [15]) for the compiler to rank the resulting compiled programs by decreasing likelihood of efficiency, without any recourse to experiments. Indeed, depending on the multiplicities and other constraints on a relation, certain representations thereof can be shown to be better than others, under certain assumptions on the targeted solver, and this either theoretically (see for instance [33] for bijections and [15] for injections) or empirically (see for instance [28] for bijections). We envisage a hybrid interactive/heuristic compiler.

Our ultimate aim is of course to design an actual *solver for relational constraints*, without going through compilation.

3 Benefits of Relational Modelling

In our experience, and as demonstrated in Section 2.3, a relational constraint modelling language leads to more *concise and intuitive models*, as well as to more *efficient and effective model formulation and verification*. Due to ESRA being *smaller* than conventional constraint programming languages, we believe it is easier to learn and master, making it a good candidate for a teaching medium. All this could entail a better dissemination of constraint technology.

Relational languages seem a good trade-off between generality and specificity, enabling *efficient solving* despite more generality. Relations are a *single*, powerful concept for elegantly modelling many aspects of combinatorial problems. Also, there are *not too many* different, and even *standard*, ways of representing relations and relational expressions. Known and future modelling insights, such as those in [15, 28, 33], can be built into the compilers, so that even time-pressed or less competent modellers can benefit from them. Modelling is unencumbered by early if not uninformed commitments to representation choices. Low-level

modelling devices such as reification and higher-order constraints can be encapsulated as implementation devices. The number of decision variables being reduced, there is even hope that directly solving the constraints at the high relational level can be faster than solving their compiled lower-level counterparts. All this illustrates that more generality need not mean poorer performance.

Relational models are more amenable to *maintenance* when the combinatorial problem changes, because most of the tedium is taken care of by the compiler. Model maintenance at the relational level reduces to adapting to the new problem, with all representation (and solving) issues left to the compiler. Very little work is involved here when a multiplicity change entails a preferable representation change for a relation. Maintenance can even be necessary when the statistical distribution of the problem instances that are to be solved changes [22]. If information on the new distribution is given to the envisaged compiler, a simple recompilation will take care of the maintenance.

Relational models are at a more suitable level for possibly automated model *reformulation*, such as via the inference and selection of suitable *implied constraints*, with again the compiler assisting in the more mundane aspects. In the BIBD and Social Golfers examples, we have observed that multiplicities provide a nice framework for discovering and stating some implied constraints. Indeed, the language makes the modeller think about making these multiplicities explicit, even if they were not in the original problem formulation.

Relational models are more amenable to *constraint analysis*. Detected properties as well as properties consciously introduced during compilation into lower-level programs, such as symmetry or bijectiveness, can then be taken into account during compilation [10], especially using tractability results [32].

There would be further benefits to an abstract modelling language if it were adopted as a *standard front-end language* for solvers. Models and instance data would then be *solver-independent* and could be shared between solvers, whatever their technology. Indeed, the targeted solvers need not even use constraint technology, but could just as well use answer-set programming, linear programming, local search, or propositional satisfiability technology, or any hybrid thereof. This would facilitate fair and homogeneous comparisons, say via new standard benchmarks, as well as foster competition in fine-tuning the compilers.

4 Conclusion

We have argued that solver-independent, abstract constraint modelling leads to a simpler and smaller language; to more concise, intuitive, and analysable models; as well as to more efficient and effective model formulation, maintenance, reformulation, and verification. All this can be achieved without sacrificing the possibility of efficient solving, so that even time-pressed or less competent modellers can be well assisted. Towards this, we have proposed the ESRA relational modelling language, showcased its elegance on some well-known problems, and outlined a compilation philosophy for such languages. To conclude, let us look at related work (Section 4.1) and future work (Section 4.2).

4.1 Related Work

We have here generalised and re-engineered our own work [11, 36, 15] on a predecessor of ESRA, now called Functional-ESRA, that only supports functional decision variables, by pursuing the aim of relational modelling outlined in [9]. Elsewhere, such ideas have recently inspired a related project [3], incorporating partition decision variables. Constraints for bag decision variables [2, 7, 34] and sequence decision variables [2, 26] have also been proposed.

This research owes a lot to previous work on relational modelling in formal methods and on ERA-style semantic data modelling, especially to the ALLOY object modelling language [16], which itself gained much from the Z specification notation [29] (and learned from UML/OCL how not to do it). Contrary to ERA modelling, we do not distinguish between attributes and relations.

In constraint programming, the commercial OPL [31] stands out as a medium-level modelling language and actually gave the impetus to design ESRA: see the BIBD example in Section 2.3 and consult [9] for a further comparison of elegant ESRA models with more awkward (published) OPL counterparts that do not provide all the benefits of Section 3. Other higher-level constraint modelling languages than ESRA have been proposed, such as ALICE [18], $CLP(Fun(D))$ [14], CLPS [2], CONJUNTO [13], EACL [30], $\{log\}$ [7], NCL [37], and the language of [24]. Our ESRA shares with them the quest for a practical declarative modelling language based on a strongly-typed fuller first-order logic than Horn clauses, with sequence, set, bag, functional, or even relational decision variables, while often dispensing with recursion, negation, and unbounded quantification. However, ESRA goes way beyond them, by advocating an ADT view (of relations), so that representations need not be fixed in advance, by providing an elegant notation for multiplicity constraints, and by promising intelligent compilation.

In the field of knowledge representation, answer-set programming (ASP) has recently been advocated [21] as a practical constraint solving paradigm, especially for dynamic domains such as planning. A set of (disjunctive) function-free clauses, where classical negation and negation as failure are allowed, is interpreted as a constraint, stating when an atom is in a solution, called an answer set or a stable model. This non-monotonic approach differs from constraint (logic) programming, where statements are used to add atomic constraints on decision variables to a constraint store, whereupon propagation and search are used to construct solutions. Implementation methods for computing the answer sets of ground programs have advanced significantly over recent years, possibly using propositional satisfiability (SAT) solvers. Also, effective grounding procedures have been devised for some classes of such programs with (schematic) variables. Sample ASP systems are DLV [19] and SMODELS [23]. Closely related are *ConstraintLingo* [8] and NP-SPEC [4]. The languages of these systems include useful features, such as cardinality and weight constraints, aggregate functions, and soft constraints. They have strictly more expressive power than propositional logic and traditional constraint (logic) programming/modelling languages, including ESRA. Again, our objective only is a language that is useful for elegantly modelling a large number of combinatorial problems. The cardinality constraint

dom *Cities*
cst $Distance : (Cities \times Cities) \longrightarrow \mathbb{N}$
var $Next : Cities \longrightarrow^{1} Cities$
minimise $\sum_{c:Cities} Distance(c, Next(c))$
such that $\forall (c_1 \& c_2 : Cities)\ Next^*(c_1) = c_2$

Fig. 7. A pretty-printed ESRA model for the Travelling Salesperson problem

of SMODELS is a restriction of the ESRA 'count' quantifier to interval multiplicities, as opposed to set multiplicities. Speed comparisons with SAT solvers were encouraging, but no comparison has been done yet with constraint solvers.

4.2 Future Work

Most of our future work has already been listed in Sections 2.4 and 3 about the compiler design and long-term benefits of relational modelling, such as the generation of implied constraints and the breaking of symmetries.

We have argued that our ESRA language is very small. This is mostly because we have not yet identified the need for any other operators or predicates. An exception to this is the need for *transitive closure relation constructors*. We aim at modelling the well-known Travelling Salesperson (TSP) problem as in Figure 7, where the transitive closure of the bijection *Next* on *Cities* is denoted by $Next^*$. This general mechanism avoids the introduction of an *ad hoc* 'circuit' constraint as in ALICE [18].

As we do not aim at a complete constraint modelling language, we can be very conservative in what missing features shall be added to ESRA when they are identified. Also, for manpower reasons, we do not yet propose other ADTs, say for bags or sequences, although this was originally part of our original vision (see Section 3.3 of [11]).

Our request for explicit model-level distinction between constants and decision variables may be eventually lifted, as the default is run-time initialisation: we could treat as constants any universally quantified variable that was actually initialised and treat all the others as decision variables. This requires a convincing example, though, as well as just-in-time compilation.

In [20], a type system is derived for binary relations that can be used as an input to specialised filtering algorithms. This kind of analysis can be integrated into the *relational solver* we have in mind.

Also, a *graphical language* could be developed for the data modelling, including the multiplicity constraints on relations, so that only the cost expression and the constraints would need to be textually expressed.

Finally, a *search language*, such as SALSA [17] or the one of OPL [31], but at the level of relational modelling, should be adjoined to the constraint modelling language proposed here, so that more expert modellers can express their own search heuristics.

Acknowledgements

This work is partially supported by grant 221-99-369 of VR, the Swedish Research Council, and by institutional grant IG2001-67 of STINT, the Swedish Foundation for International Cooperation in Research and Higher Education. We thank Nicolas Beldiceanu, Mats Carlsson, Esra Erdem, Brahim Hnich, Daniel Jackson, Zeynep Kızıltan, François Laburthe, Gerrit Renker, Christian Schulte, Mark Wallace, and Simon Wrang for stimulating discussions, as well as the constructive reviewers of previous versions of this paper.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. F. Ambert, B. Legeard, and E. Legros. Programmation en logique avec contraintes sur ensembles et multi-ensembles héréditairement finis. *Techniques et Sciences Informatiques*, 15(3):297–328, 1996.
3. A. Bakewell, A. M. Frisch, and I. Miguel. Towards automatic modelling of constraint satisfaction problems: A system based on compositional refinement. In *Proceedings of the 2nd International Workshop on Modelling and Reformulating CSPs*, pages 3–17, 2003. Available at `http://www-users.cs.york.ac.uk/~frisch/Reformulation/03/`.
4. M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NPSPEC: An executable specification language for solving all problems in NP. In G. Gupta, editor, *Proceedings of PADL'99*, volume 1551 of *LNCS*, pages 16–30. Springer-Verlag, 1999.
5. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. Hartel, and H. Kuchen, editors, *Proceedings of PLILP'97*, number 1292 in LNCS, pages 191–206. Springer-Verlag, 1997.
6. M. Denecker, N. Pelov, and M. Bruynooghe. Ultimate well-founded and stable semantics for logic programs with aggregates. In *Proceedings of ICLP'01*, volume 2237 of *LNCS*, pages 212–226. Springer-Verlag, 2001.
7. A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM Transactions on Programming Languages and Systems*, 22(5):861–931, 2000.
8. R. Finkel, V. Marek, and M. Truszczyński. Tabular constraint-satisfaction problems and answer-set programming. In *Proceedings of the AAAI Spring Symposium on Answer-Set Programming*, 2001. Available at `http://www.cs.nmsu.edu/~tson/ASP2001/`.
9. P. Flener. Towards relational modelling of combinatorial optimisation problems. In C. Bessière, editor, *Proceedings of the IJCAI'01 Workshop on Modelling and Solving Problems with Constraints*, pages 31–38, 2001. Available at `http://www.lirmm.fr/~bessiere/ws_ijcai01/`.
10. P. Flener, A. M. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, and T. Walsh. Matrix modelling: Exploiting common patterns in constraint programming. In *Proc. of the 1st Int'l Workshop on Reformulating CSPs*, pages 27–41, 2002. Available at `http://www-users.cs.york.ac.uk/~frisch/Reformulation/02/`.
11. P. Flener, B. Hnich, and Z. Kızıltan. Compiling high-level type constructors in constraint programming. In I. Ramakrishnan, editor, *Proceedings of PADL'01*, volume 1990 of *LNCS*, pages 229–244. Springer-Verlag, 2001.

12. P. Flener, J. Pearson, and M. Ågren. The Syntax, Semantics, and Type System of esra. Technical report, ASTRA group, April 2003. Available at `http://www.it.uu.se/research/group/astra/`.
13. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
14. T. J. Hickey. Functional constraints in CLP languages. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 355–381. The MIT Press, 1993.
15. B. Hnich. *Function Variables for Constraint Programming*. PhD thesis, Department of Information Science, Uppsala University, Sweden, 2003. Available at `http://publications.uu.se/theses/`.
16. D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. *Software Engineering Notes*, 26(5):62–73, 2001. Proceedings of FSE/ESEC'01.
17. F. Laburthe and Y. Caseau. SALSA: A language for search algorithms. *Constraints*, 7:255–288, 2002.
18. J.-L. Laurière. A language and a program for stating and solving combinatorial problems. *Artificial Intelligence*, 10(1):29–127, 1978.
19. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. In *ACM Transactions on Computational Logic*, forthcoming. Available at `http://arxiv.org/ps/cs.AI/0211004`.
20. D. Lesaint. Inferring constraint types in constraint programming. In P. Van Hentenryck, editor, *Proceedings of CP'02*, volume 2470 of *LNCS*, pages 492–507. Springer-Verlag, 2002.
21. V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138:39–54, 2002.
22. S. Minton. Automatically configuring constraint satisfaction programs: A case study. *Constraints*, 1(1–2):7–43, 1996.
23. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and AI*, 25(3–4):241–273, 1999.
24. N. Pelov and M. Bruynooghe. Extending constraint logic programming with open functions. In *Proceedings of PPDP'00*, pages 235–244. ACM Press, 2000.
25. N. Pelov, M. Denecker, and M. Bruynooghe. Partial stable models for logic programs with aggregates. In *Proceedings of LPNMR'04*, volume 2923 of *LNCS*, pages 207–219. Springer-Verlag, 2004.
26. G. Pesant. A regular language membership constraint for sequences of variables. In *Proceedings of the 2nd International Workshop on Modelling and Reformulating CSPs*, pages 110–119, 2003. Available at `http://www-users.cs.york.ac.uk/~frisch/Reformulation/03/`.
27. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
28. B. M. Smith. Modelling a permutation problem. Technical Report 18, School of Computing, University of Leeds, UK, 2000. Also in *Proceedings of the ECAI'00 Workshop on Modelling and Solving Problems with Constraints*.
29. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
30. E. Tsang, P. Mills, R. Williams, J. Ford, and J. Borrett. A computer-aided constraint programming system. In J. Little, editor, *Proceedings of PACLP'99*, pages 81–93. The Practical Application Company, 1999.
31. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

32. P. Van Hentenryck, P. Flener, J. Pearson, and M. Ågren. Tractable symmetry breaking for CSPs with interchangeable values. In *Proceedings of IJCAI'03*, pages 277–282. Morgan Kaufmann, 2003.
33. T. Walsh. Permutation problems and channelling constraints. In R. Nieuwenhuis and A. Voronkov, editors, *Proc. of LPAR'01*, number 2250 in LNCS, pages 377–391. Springer-Verlag, 2001.
34. T. Walsh. Consistency and propagation with multiset constraints: A formal viewpoint. In F. Rossi, editor, *Proceedings of CP'03*, number 2833 in LNCS, pages 724–738. Springer-Verlag, 2003.
35. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1999.
36. S. Wrang. Implementation of the ESRA Constraint Modelling Language. Master's thesis, Master's Thesis in Computing Science 223, Department of Information Technology, Uppsala University, Sweden, 2002. Available at `ftp://ftp.csd.uu.se/pub/papers/masters-theses/`.
37. J. Zhou. Introduction to the constraint language NCL. *Journal of Logic Programming*, 45(1–3):71–103, 2000.

Paper B

Set Variables and Local Search

Reprinted from:

R. Barták and M. Milano (editors), Proceedings of the 2nd International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - CP-AI-OR 2005, pp. 19–33, Lecture Notes in Computer Science, volume 3524, Springer-Verlag, 2005.

Set Variables and Local Search*

Magnus Ågren, Pierre Flener, and Justin Pearson

Department of Information Technology,
Uppsala University, Box 337,
SE – 751 05 Uppsala, Sweden
{agren, pierref, justin}@it.uu.se

Abstract. Many combinatorial (optimisation) problems have natural models based on, or including, set variables and set constraints. This was already known to the constraint programming community, and solvers based on constructive search for set variables have been around for a long time. In this paper, set variables and set constraints are put into a local-search framework, where concepts such as configurations, penalties, and neighbourhood functions are dealt with generically. This scheme is then used to define the penalty functions for five (global) set constraints, and to model and solve two well-known applications.

1 Introduction

Many combinatorial (optimisation) problems have natural models based on, or including, set variables and set constraints. Classical examples include set partitioning and set covering, and such problems also occur as sub-problems in many real-life applications, such as airline crew rostering, tournament scheduling, time-tabling, and nurse rostering. This was already known to the constraint programming community, and constructive search (complete) solvers for set variables have been around for a long time now (see for example [11, 15, 19, 2]).

Complementary to constructive search, local search [1] is another common technique for solving combinatorial (optimisation) problems. Although not complete, it usually scales very well to large problem instances and often compares well to, or outperforms, other techniques. Historically, the constraint programming community has been mostly focused on constructive search and has only recently started to apply its ideas to local search. This means that concepts such as high declarativeness, global constraints with underlying incremental algorithms, and high-level modelling languages for local search have been introduced there (see [12, 25, 22, 16, 10, 7, 13, 23, 14, 6] for instance).

In this paper, we introduce set variables and (global) set constraints to constraint-based local search. More specifically, our *contributions* are as follows:

– We put the local-search concepts of penalties, configurations, and neighbourhood functions into a *set-variable framework*. (Section 2)

* This paper significantly extends and revises Technical Report 2004-015 of the Department of Information Technology, Uppsala University, Sweden.

R. Barták and M. Milano (Eds.): CPAIOR 2005, LNCS 3524, pp. 19–33, 2005.

- In order to be able to use (global) set constraints generally in local search, we propose a *generic penalty scheme*. We use it to give the *penalty definitions of five (global) set constraints*. Other than their well-known *modelling* merits, we show that (global) set constraints provide opportunities for a hardwired global reasoning while *solving*, which would otherwise have to be hand-coded each time for lower-level encodings of set variables, such as integer variables for the characteristic functions of their set values. (Section 3)
- In order to obtain efficient solution algorithms, we propose methods for the *incremental penalty maintenance* of the (global) set constraints. (Section 4)
- The (global) set constraints are used to *model and solve two well-known problems*, with promising results that motivate further research. (Section 5)

After this, Section 6 discusses related and future work and concludes the paper.

2 Local Search on (Set) Constraint Satisfaction Problems

A *constraint satisfaction problem (CSP)* is a triple $\langle V, D, C\rangle$, where V is a finite set of variables, D is a finite set of finite domains, each $D_v \in D$ containing the set of possible values for the corresponding variable $v \in V$, and C is a finite set of constraints, each $c \in C$ being defined on a subset of the variables in V and specifying their valid combinations of values.

The definition above is very general and may be used with any choice of finite-domain variables. The variables in V may, for example, range over sets of integers (integer variables), strings, or, as in our case, sets of values of some type (set variables, defined formally below). Of course, a CSP may also contain variables with several kinds of domains. As an example, consider a CSP $\langle V, D, C\rangle$ in which some variables $\{i_1, \ldots, i_k\} \subset V$ are integer variables, and some other variables $\{s_1, \ldots, s_k\} \subset V$ are set variables. These could for instance be connected with constraints stating that the cardinality of each s_j must not exceed i_j.

In this paper, we assume that *all* the variables are set variables, and that all the constraints are stated on variables of this kind. This is of course a limitation, since many models contain both set variables and integer variables. However, mixing integer variables and set variables makes the constraints harder to define, and we consider this to be future work. Fortunately, interesting applications, such as the two in this paper, are already possible to model.

Definition 1 (Set Variable and its Universe). *Let $P = \langle V, D, C\rangle$ be a CSP. A variable $s \in V$ is a* set variable *if its corresponding domain $D_s = 2^{U_s}$, where U_s is a finite set of values of some type, called the* universe *of s.*

Note that this definition does not allow the indication of a non-empty set of required values in the universe of a set variable, hence this must be done here by an explicit constraint. This is left as future work, as not necessary for our present purpose.

Definition 2 (Configuration). *Let $P = \langle V, D, C\rangle$ be a CSP. A* configuration *for P is a total function $k : V \to \bigcup_{s \in V} D_s$ such that $k(s) \in D_s$ for all $s \in V$.*

Definition 3 (Delta of Configurations). *Let $P = \langle V, D, C\rangle$ be a CSP and let k and k' be two configurations for P. The* delta of k and k'*, denoted $delta(k, k')$, is the set $\{(s, v, v') \mid s \in V \ \&\ v = k(s) - k'(s) \ \&\ v' = k'(s) - k(s) \ \&\ v \neq v'\}$, where $-$ stands for the set difference.*

Example 1. Consider a CSP $P = \langle\{s_1, s_2, s_3\}, \{D_{s_1}, D_{s_2}, D_{s_3}\}, C\rangle$ where $D_{s_1} = D_{s_2} = D_{s_3} = 2^{\{d_1,d_2,d_3\}}$ (hence $U_{s_1} = U_{s_2} = U_{s_3} = \{d_1, d_2, d_3\}$). One possible configuration for P is defined as $k(s_1) = \{d_3\}, k(s_2) = \{d_1, d_2\}, k(s_3) = \emptyset$, or equivalently as the set of mappings $\{s_1 \mapsto \{d_3\}, s_2 \mapsto \{d_1, d_2\}, s_3 \mapsto \emptyset\}$. Another configuration for P is defined as $k' = \{s_1 \mapsto \emptyset, s_2 \mapsto \{d_1, d_2, d_3\}, s_3 \mapsto \emptyset\}$. Now, the delta of k and k' is $delta(k, k') = \{(s_1, \{d_3\}, \emptyset), (s_2, \emptyset, \{d_3\})\}$.

Definition 4 (Neighbourhood Function). *Let K denote the set of all possible configurations for a CSP P and let $k \in K$. A* neighbourhood function *for P is a function $\mathcal{N} : K \to 2^K$. The* neighbourhood *of P with respect to k and $\mathcal{N}$ is the set of configurations $\mathcal{N}(k)$.*

Example 2. Consider P and k from Example 1. A possible neighbourhood of P with respect to k and some neighbourhood function $\mathcal{N}$ for P is the set $\mathcal{N}(k) = \{k_1 = \{s_1 \mapsto \emptyset, s_2 \mapsto \{d_1, d_2, d_3\}, s_3 \mapsto \emptyset\}, k_2 = \{s_1 \mapsto \emptyset, s_2 \mapsto \{d_1, d_2\}, s_3 \mapsto \{d_3\}\}$. This neighbourhood function moves the value d_3 in s_1 to variable s_2 or variable s_3, decreasing the cardinality of s_1 and increasing the one of s_2 or s_3.

We will use two *general neighbourhoods* in this paper, which are defined next. For both, let $s \in V$, $S \subseteq V - \{s\}$, and let $k \in K$ be a configuration for a CSP $P = \langle V, D, C\rangle$, where K is the set of all configurations for P. The first one, called *move*, is defined by the neighbourhood function with the same name:

$$move(s, S)(k) = \{k' \in K \mid \exists d \in k(s) : s' \in S \ \&\ d \in U_{s'} - k(s') \ \&\ delta(k, k') = \{(s, \{d\}, \emptyset), (s', \emptyset, \{d\})\}\}$$

This neighbourhood, given k, is the set of all neighbourhoods k' that differ from k in the definition of two distinct set variables s and s', the difference being that there exists exactly one $d \in k(s)$ such that $d \in k(s) \Leftrightarrow d \notin k'(s)$ and $d \notin k(s') \Leftrightarrow d \in k'(s')$. Hence, d was moved from s to s'.

The second one, called *swap*, is defined by the neighbourhood function:

$$swap(s, S)(k) = \{k' \in K \mid \exists d \in k(s) : \exists d' \in U_s - k(s) : s' \in S \ \&\ d' \in k(s') \ \&\ d \in U_{s'} - k(s') \ \&\ delta(k, k') = \{(s, \{d\}, \{d'\}), (s', \{d'\}, \{d\})\}\}$$

This neighbourhood, given k, is the set of all neighbourhoods k' that differ from k in the definition of two distinct set variables s and s', the difference being that there exists exactly one pair ($d \in k(s)$, $d' \in U_s - k(s)$) such that $d \in k(s) \Leftrightarrow d \notin k'(s)$ and $d \notin k(s') \Leftrightarrow d \in k'(s')$, and the opposite for d'. Hence, d and d' were swapped between s and s'.

We will now define the notion of penalty of a constraint, which, informally, is an estimate on how much a constraint is violated. Below is a general definition, followed by a generic scheme for balancing the penalties of different constraints, which is then specialised for each constraint in Section 3.

Definition 5 (Penalty). *Let $P = \langle V, D, C \rangle$ be a CSP and let K denote the set of all possible configurations for P. A* penalty of a constraint $c \in C$ *is a function $penalty(c) : K \to \mathbb{N}$. The* penalty of P with respect to k *is the sum $\sum_{c \in C} penalty(c)(k)$.*

Example 3. Consider once again P from Example 1 and let c_1 and c_2 be the constraints $s_1 \subseteq s_2$ and $d_3 \in s_3$ respectively. Let the penalty functions of c_1 and c_2 be defined as: $penalty(c_1)(k) = |k(s_1) - k(s_2)|$, and $penalty(c_2)(k) = 0$, *if* $d_3 \in k(s_3)$, *or* 1, *otherwise* . Now, the penalties of P with respect to the different configurations in the neighbourhood of Example 2 are $penalty(c_1)(k_1) + penalty(c_2)(k_1) = 1$, and $penalty(c_1)(k_2) + penalty(c_2)(k_2) = 0$ respectively.

In order for a constraint-based local-search approach to be effective, different constraints should have balanced penalty definitions [6]: i.e. for a set of constraints C, no $c \in C$ should be easier in general to satisfy compared to any other $c' \in C$. This may be application dependent, in which case weights could be added to tune the penalties, see [13] for example. For set constraints, we believe that one such penalty definition is to let (by extension of the integer-variable ideas in [10]) the penalty of a set constraint c be the length of the shortest sequence of atomic set operations (defined below) that must be performed on the variables in c under a configuration k in order to satisfy c.

Definition 6 (Atomic Set Operations). *Let $P = \langle V, D, C \rangle$ be a CSP, let k be a configuration for P, and let $s \in V$. An* atomic set operation on $k(s)$ *is one of the following changes to $k(s)$:*

1. *Add a value d to $k(s)$ from its complement $U_s - k(s)$, denoted $Add(k(s), d)$.*
2. *Remove a value d from $k(s)$, denoted $Remove(k(s), d)$.*

Note that no value-replacement operation is considered here; its inclusion would imply a reduction of some of the penalties in Section 3.

Example 4. Performing $\Delta = [Add(k(s), d),\ Remove(k(s), b),\ Add(k(s'), b)]$ on $k(s) = \{a, b, c\}$ and $k(s') = \emptyset$ will yield $\Delta(k(s)) = \{a, c, d\}$ and $\Delta(k(s')) = \{b\}$.

Definition 7 (Operation-Based Penalty for Set Constraints). *Let $P = \langle V, D, C \rangle$ be a CSP and let K be the set of all configurations for P. Let $c \in C$ be a constraint defined on a set of set variables $S \subseteq V$. The* penalty of c, *$penalty(c) : K \to \mathbb{N}$, is the length of the shortest sequence of atomic set operations that must be performed in order to satisfy c given a specific configuration k.*

From this definition it follows that $penalty(c)(k) = 0$ if and only if c is satisfied with respect to k. Also, as will be seen, to find a penalty that complies with this definition for a given set constraint is not always obvious.

3 (Global) Set Constraints and Their Penalties

We now present five (global) set constraints and define their penalties. Throughout this section, we assume that k is a configuration for a CSP $P = \langle V, D, C \rangle$, and that $c \in C$.

3.1 AllDisjoint

The global constraint $\mathit{AllDisjoint}(S)$, where $S = \{s_1, \ldots, s_n\}$ is a set of set variables, expresses that all distinct pairs in S are disjoint, i.e. that $\forall i < j \in 1 \ldots n : s_i \cap s_j = \emptyset$. The penalty of an $\mathit{AllDisjoint}(S)$ constraint under k is equal to the length of the shortest sequence Δ of atomic set operations of the form $\mathit{Remove}(k(s), d)$ that must be performed in order for $\forall i < j \in 1 \ldots n : \Delta(k(s_i)) \cap \Delta(k(s_j)) = \emptyset$ to hold. We define the penalty as:

$$\mathit{penalty}(\mathit{AllDisjoint}(S))(k) = \left(\sum_{s \in S} |k(s)|\right) - \left|\bigcup_{s \in S} k(s)\right| \tag{1}$$

Indeed, we need to remove all repeated occurrences of any value, and their number equals the difference between the sum of the set sizes and the size of their union. Hence the following proposition:

Proposition 1. *The penalty (1) is correct with respect to Definition 7.*

3.2 Cardinality

The constraint $\mathit{Cardinality}(s, m)$, where s is a set variable and m a natural-number constant, expresses that the cardinality of s is equal to m, i.e. that $|s| - m$. This constraint would of course be more powerful if we allowed m to be an integer variable. However, as was mentioned earlier, the penalty would be more complicated if we did this, and we see this as future work.

The penalty of a $\mathit{Cardinality}(s, m)$ constraint under k is equal to the length of the shortest sequence Δ of atomic set operations of the form $\mathit{Add}(k(s), d)$ or $\mathit{Remove}(k(s), d)$ that must be performed in order for $|\Delta(k(s))| = m$ to hold. The penalty below expresses this:

$$\mathit{penalty}(\mathit{Cardinality}(s, m))(k) = \mathit{abs}(|k(s)| - m) \tag{2}$$

where $\mathit{abs}(e)$ denotes the absolute value of the expression e. Indeed, we need to add (remove) exactly as many values to (from) $k(s)$ in order to increase (decrease) its cardinality to m. Hence the following proposition:

Proposition 2. *The penalty (2) is correct with respect to Definition 7.*

3.3 MaxIntersect

The global constraint $\mathit{MaxIntersect}(S, m)$, where $S = \{s_1, \ldots, s_n\}$ is a set of set variables and m a natural-number constant, expresses that the cardinality of the intersection between any distinct pair in S is at most m, i.e. that $\forall i < j \in 1 \ldots n : |s_i \cap s_j| \leq m$. This constraint expresses the same as an $\mathit{AllDisjoint}(S)$ constraint when $m = 0$. However, as will be seen, keeping the *AllDisjoint* constraint is useful for this special case. Again, allowing m to be an integer variable would make the constraint more powerful and is future work.

The penalty of a $\mathit{MaxIntersect}(S, m)$ constraint under k is equal to the length of the shortest sequence Δ of atomic set operations of the form $\mathit{Remove}(k(s), d)$ that must be performed such that $\forall i < j \in 1 \dots n : |\Delta(k(s_i)) \cap \Delta(k(s_j))| \leq m$ holds. In fact, finding a closed form for the exact penalty of a *MaxIntersect* constraint with respect to Definition 7 turns out not to be that easy. The following expression gives an upper bound on this penalty, namely the sum of the excesses of the intersection sizes:

$$\mathit{penalty}(\mathit{MaxIntersect}(S, m))(k) \leq \sum_{1 \leq i < j \leq n} \mathit{max}(|k(s_i) \cap k(s_j)| - m, 0) \qquad (3)$$

Example 5. Assume that $k(s_1) = \{d_1, d_2, d_3\}$, $k(s_2) = \{d_2, d_3, d_4\}$, $k(s_3) = \{d_1, d_3, d_4\}$, and that $c = \mathit{MaxIntersect}(\{s_1, s_2, s_3\}, 1)$. The penalty of c according to (3) is $2 + 2 + 2 = 3$. Indeed, we may satisfy c by performing the sequence of 3 operations $[\mathit{Remove}(k(s_1), d_1), \mathit{Remove}(k(s_2), d_2), \mathit{Remove}(k(s_3), d_3)]$. However, this is not the shortest sequence that achieves this, since after performing $[\mathit{Remove}(k(s_1), d_3), \mathit{Remove}(k(s_2), d_3)]$, the constraint c is also satisfied.

Proposition 3. *The bound of (3) is an optimal upper bound w.r.t. Definition 7.*

Proposition 4. *The upper bound of (3) is zero iff $\mathit{MaxIntersect}(S, m)$ holds.*

However, the upper bound of (3) is not correct with respect to Definition 7 when $m = 0$. Consider $s_1 = \{d_1, d_2\}$, $s_2 = \{d_2, d_3\}$, and $s_3 = \{d_2, d_3\}$. The penalty under (3) of $\mathit{MaxIntersect}(\{s_1, s_2, s_3\}, 0)$ is $1 + 1 + 2 = 4$ whereas the one of $\mathit{AllDisjoint}(\{s_1, s_2, s_3\})$ correctly is $6 - 3 = 3$ under (1).

We may also obtain a lower bound, by using a lemma due to Corrádi [8].

Lemma 1 (Corrádi). *Let $s_1, \dots, s_n$ be r-element sets and U be their union. If $|s_i \cap s_j| \leq m$ for all $i \neq j$, then $|U| \geq \frac{r^2 \cdot n}{r + (n-1) \cdot m}$.*

This lemma can be applied for n ground sets that do not necessarily all have the *same* cardinality r, but rather with r being the *maximum* of their cardinalities, as is the case with $\mathit{MaxIntersect}(S, m)$ and $|S| = n$. It suffices to apply the corrective term $\delta = n \cdot r - \sum_{s \in S} |k(s)|$ when using the lower bound for a configuration k where $r = \mathit{max}_{s \in S} |k(s)|$. Note that δ is the amount of distinct new elements (from a sufficiently large fictitious universe disjoint from $\bigcup_{s \in S} k(s)$) that one must add to the sets in $\{k(s) \mid s \in S\}$ in order to make them all be of size r.

We now have the following lower bound on the penalty of a $\mathit{MaxIntersect}(S, m)$ constraint under a configuration k (where $|S| = n$ and $r = \mathit{max}_{s \in S} |k(s)|$):

$$\mathit{penalty}(\mathit{MaxIntersect}(S, m))(k) \geq \left\lceil \frac{r^2 \cdot n}{r + (n-1) \cdot m} \right\rceil - \left(n \cdot r - \sum_{s \in S} |k(s)| \right) - \left| \bigcup_{s \in S} k(s) \right| \qquad (4)$$

Example 6. Recall Example 5, where $m = 1$ and the $n = 3$ sets are of the same size $r = 3$, hence $\delta = 0$, and have a union of 4 elements. We get $\mathit{penalty}(c)(k) \geq \lceil \frac{27}{5} \rceil - 0 - 4 = 2$, which is correct with respect to Definition 7.

Now, the following proposition follows from Lemma 1:

Proposition 5. *The bound of (4) is an optimal lower bound w.r.t. Definition 7.*

The next proposition establishes what happens when $m = 0$, in which case $\mathit{MaxIntersect}(S, m)$ is equivalent to $\mathit{AllDisjoint}(S)$:

Proposition 6. *The lower bound of (4) is correct wrt Definition 7 when $m = 0$.*

Proof. When $m = 0$, then $\left\lceil \frac{r^2 \cdot n}{r+(n-1)\cdot m} \right\rceil = r \cdot n$ and the lower bound of (4) simplifies into the penalty expression (1). Hence it is correct, by Proposition 1.

Unfortunately, the lower bound is sometimes zero even though the constraint is violated. Consider $n = 10$ sets, all of size $r = 3$ (hence $\delta = 0$), that should have pairwise intersections of at most $m = 1$ element and that have a union of 8 elements. Then (4) gives 0 as lower bound on the penalty, but the constraint is violated as there are no such 10 sets, hence m would have to be at least 2.

However, we may still use (4) for the *MaxIntersect* constraint, but it would have to be in conjunction with (3), with the condition that if the lower bound of (4) is zero, then one uses the upper bound of (3) instead. In our experience, the lower bound of (4) is frequently correct. This also argues for keeping the explicit constraint *AllDisjoint*, since for that constraint (4) gives the correct penalty.

An often tighter upper bound than the one of (3) can be obtained by Algorithm 1. It obtains an estimate of the penalty by returning the length of a sequence of atomic set operations constructed in the following way: (i) Start with the empty sequence. (ii) Until the constraint is satisfied, add an atomic set operation removing a value that belongs to a set variable that takes part in the largest number of violating intersections. The algorithm uses the upper bound of (3) as the exit criterion, as it is zero only upon satisfaction of the constraint, by Proposition 4.

Algorithm 1 Calculating the penalty of a *MaxIntersect* constraint

function $\mathit{penalty\_max\_intersect}(S, m)(k)$
  $l \leftarrow 0$
  while $\mathit{penalty}(\mathit{MaxIntersect}(S, m))(k) > 0$ **do** ▷ According to (3)
    choose $d \in \bigcup_{s \in S} k(s)$ s.t. $|\{(i, j) \mid i < j \ \ \& \ \ d \in k(s_i) \cap k(s_j) \ \ \& \ \ |k(s_i) \cap k(s_j)| > m\}|$ is maximised.
    choose $s_i \in S$ s.t. $|\{s_j \in S \mid i \neq j \ \ \& \ \ d \in k(s_i) \cap k(s_j) \ \ \& \ \ |k(s_i) \cap k(s_j)| > m\}|$ is maximised.
    $l \leftarrow l + 1$ ▷ i.e. an imaginary $\mathit{Remove}(k(s_i), d)$ operation was added
    Replace the binding for s_i in k by $s_i \mapsto k(s_i) - \{d\}$
  return l

In the current implementation of the *MaxIntersect* constraint, we use the upper bound given by (3). As we have seen, this is not always a good estimate on the penalty with respect to Definition 7. In the future, we plan to use (4) in conjunction with (3) or (an incremental variant of) Algorithm 1.

3.4 MaxWeightedSum

The constraint $\mathit{MaxWeightedSum}(s, w, m)$, where s is a set variable, $w : U_s \to \mathbb{N}$ is a weight function from the universe of s to the natural numbers, and m is a natural-number constant, expresses that $\sum_{d \in s} w(d) \leq m$. Note that we do not allow negative weights nor m to be an integer variable. Allowing these would need a redefinition of the penalty below.

The penalty of a $\mathit{MaxWeightedSum}(s, w, m)$ constraint under k is equal to the length of the shortest sequence Δ of operations of the form $\mathit{Remove}(k(s), d)$ that must be performed in order for $\sum_{d \in \Delta(k(s))} w(d) \leq m$ to hold. We define the following penalty:

$$\begin{aligned} &\mathit{penalty}(\mathit{MaxWeightedSum}(s, w, m))(k) = \\ &\qquad \mathit{min\_card}\left(\left\{s' \subseteq k(s) \mid \textstyle\sum_{d' \in s'} w(d') \geq \left(\sum_{d \in k(s)} w(d)\right) - m\right\}\right) \end{aligned} \tag{5}$$

where $\mathit{min\_card}(Q)$ denotes the cardinality of a set $q \in Q$ such that for all $q' \in Q$, $|q| \leq |q'|$, or 0 if $Q = \emptyset$. Indeed, we must remove at least the smallest set of values from $k(s)$ such that their weighted sum is at least the difference between the weighted sum of all values in $k(s)$ and m. Hence the following proposition:

Proposition 7. *The penalty (5) is correct with respect to Definition 7.*

3.5 Partition

The global constraint $\mathit{Partition}(S, q)$, where $S = \{s_1, \ldots, s_n\}$ is a set of set variables and q is a ground set of values, expresses that the variables in S are all disjoint, i.e. that $\forall i < j \in 1 \ldots n : s_i \cap s_j = \emptyset$, and that their union is equal to q, i.e. that $\bigcup_{s \in S} s = q$. Note that this definition of a partition allows one or more variables in S to be empty, which is useful in some applications, such as the progressive party problem below. The set q, called the *reference set*, could be generalised to be a set variable. The applications we currently look at do not expect this but this may change in the future. In that case, the penalty function below would have to be changed to take this into account.

The penalty of a $\mathit{Partition}(S, q)$ constraint under k is equal to the length of the shortest sequence Δ of atomic set operations that must be performed in order for $\forall i < j \in 1 \ldots n : \Delta(k(s_i)) \cap \Delta(k(s_j)) = \emptyset \;\;\&\;\; \bigcup_{s \in S} \Delta(k(s)) = q$ to hold. The following penalty expresses this:

$$\mathit{penalty}(\mathit{Partition}(S, q))(k) = \left(\sum_{s \in S} |k(s)|\right) - \left|\bigcup_{s \in S} k(s)\right| + \left|q - \bigcup_{s \in S} k(s)\right| \tag{6}$$

Indeed, the first two terms are those in (1) for *AllDisjoint* and the third term expresses that all unused elements of the reference set must be added to some set of the partition for the union to hold. Hence the following proposition:

Proposition 8. *The penalty (6) is correct with respect to Definition 7.*

Note that this penalty could be reduced by allowing replacement operations.

4 Incrementally Maintaining Penalties

This section presents how the penalties are maintained for two of the presented constraints, *AllDisjoint* and *MaxIntersect*. For the other three, *Partition* is similar to *AllDisjoint*, while *Cardinality* and *MaxWeightedSum* are rather straightforward to maintain. Since in local search one may need to perform many iterations, and since each iteration usually requires searching through a large neighbourhood, it is crucial that the penalty of a neighbouring configuration is computed efficiently. In order to do this, it is important to use *incremental algorithms* that, given a current configuration k, do not recompute from scratch the penalty of a neighbouring configuration k', but rather compute the penalty with respect to the penalty of k and the difference between k and k'.

This technique is used, for instance, in [12, 22] where invariants are used to get efficient incremental algorithms from high-level, declarative descriptions. In this paper, the incrementality is achieved explicitly for each constraint, and we consider it to be future work to implement this in a more general and elegant way. The aim of this paper is to explore the usefulness of the proposed framework and penalty definitions for set constraints.

4.1 Incrementally Maintaining *AllDisjoint*

Recall the penalty (1) for an *AllDisjoint* constraint in Section 3.1. In order to maintain this incrementally, we use a table *count* of integers, indexed by the values in $U = \bigcup_{s \in S} U_s$, such that $count[d]$ is equal to the number of variables that contain d. Now, the sum in (1) is equal to $\sum_{d \in U}(count[d]-1)$ as it suffices to remove a value $d \in \bigcup_{s \in S} k(s)$ from all but one of the set variables in $\{s \in S \mid d \in k(s)\}$ in order to satisfy the constraint. This is easy to maintain incrementally given an atomic set operation.

4.2 Incrementally Maintaining *MaxIntersect*

Recall the penalty bound of (3) for a *MaxIntersect* constraint. In order to maintain this incrementally, we use the following two data structures: (i) A table *variables* indexed by the values in $U = \bigcup_{s \in S} U_s$, such that $variables[d]$ is the set of variables that d is a member of; (ii) for each variable s_i, a table $s_i.intersects$ indexed by the values in $\{i+1, \ldots, n\}$ such that $s_i.intersects[j] = |k(s_i) \cap k(s_j)|$.

The sum in (3) is then equal to $\sum_{1 \leq i < j \leq n} max(s_i.intersects[j] - m, 0)$ and all this may be maintained incrementally in the following way, given an atomic set operation o. If $o = Add(k(s_i), d)$ then (i) add s_i to $variables[d]$; (ii) for each variable s_j in $variables[d]$ such that $j > i$: if $s_i.intersects[j] \geq m$ then increase the sum in (3) by 1; and (iii) for each variable s_j in $variables[d]$ such that $j > i$: increase $s_i.intersects[j]$ by 1. If $o = Remove(k(s_i), d)$ then (i) remove s_i from $variables[d]$; (ii) for each variable s_j in $variables[d]$ such that $j > i$: if $s_i.intersects[j] > m$ then decrease the sum in (3) by 1; and (iii) for each variable s_j in $variables[d]$ such that $j > i$: decrease $s_i.intersects[j]$ by 1.

Implementing these ideas with respect to the lower bound of (4) and Algorithm 1 is future work.

5 Applications

This section presents two well-known applications for constraint programming: the *Progressive Party Problem* and the *Social Golfers Problem*. They both have natural models based on set variables. They have previously been solved both using constructive and local search. See, for instance, the references [21, 10, 25, 13, 6, 24] and [3, 20, 18, 9], respectively. The constraints in Section 3 as well as the search algorithms were implemented in OCaml and the experiments were run on an Intel 2.4 GHz Linux machine with 512 MB memory.

5.1 The Progressive Party Problem (PPP)

The problem is to timetable a party at a yacht club. Certain boats are designated as hosts, while the crews of the remaining boats are designated as guests. The crew of a host boat remains on board throughout the party to act as hosts, while the crew of a guest boat together visits host boats over a number of periods. The crew of a guest boat must party at some host boat each period (constraint c_1). The spare capacity of any host boat is not to be exceeded at any period by the sum of the crew sizes of all the guest boats that are scheduled to visit it then (constraint c_2). Any guest crew can visit any host boat in at most one period (constraint c_3). Any two distinct guest crews can visit the same host boat in at most one period (constraint c_4).

A Set-Based Model. Let H be the set of host boats and let G be the set of guest boats. Furthermore, let $\mathit{capacity}(h)$ and $\mathit{size}(g)$ denote the spare capacity of host boat h and the crew size of guest boat g, respectively. Let *periods* be the number of periods we want to find a schedule for and let P be the set $\{1, \ldots, \mathit{periods}\}$. Now, let $s_{(h,p)}$, where $h \in H$ and $p \in P$, be a set variable containing the set of guest boats whose crews boat h hosts during period p. Then the following constraints model the problem:

$(c_1) : \forall p \in P : \mathit{Partition}(\{s_{(h,p)} \mid h \in H\}, G)$
$(c_2) : \forall h \in H : \forall p \in P : \mathit{MaxWeightedSum}(s_{(h,p)}, \mathit{size}, \mathit{capacity}(h))$
$(c_3) : \forall h \in H : \mathit{AllDisjoint}(\{s_{(h,p)} \mid p \in P\})$
$(c_4) : \mathit{MaxIntersect}(\{s_{(h,p)} \mid h \in H \ \ \& \ \ p \in P\}, 1)$

Solving The PPP. If we are careful when defining an initial configuration and a neighbourhood for the PPP, we may be able to exclude some of its constraints. For instance, it is possible to give the variables $s_{(h,p)}$ an initial configuration and a neighbourhood that respect c_1. We can do this (i) by assigning random disjoint subsets of G to each $s_{(h,p)}$, where $h \in H$, for each period $p \in P$, making sure that each $g \in G$ is assigned to some $s_{(h,p)}$ and (ii) by using a neighbourhood specifying that guests from a host boat h are moved to another host boat h' in the same period, and nothing else.

Algorithm 2 is the solving algorithm we used for the PPP. It takes the constant sets P, G, H, and the functions *capacity* and *size* as defined above as

parameters, specifying an instance of the PPP, and returns a configuration k for a CSP with respect to that instance. *MaxIter* and *MaxNonImproving* are additional arguments as described below. If $penalty(\langle V, D, C\rangle)(k) = 0$, then a solution was found within *MaxIter* iterations. The algorithm uses the notion of *conflict of a variable* (line 10), which, informally, is an estimate on how much a variable contributes to the total penalty of a set of constraints with respect to a configuration.

Algorithm 2 Solving the PPP

```
 1: procedure solve_progressive_party(P, G, H, capacity, size)
 2:     Initialise ⟨V, D, C⟩ w.r.t. P, G, H, capacity, and size to be a CSP ∈ PPP
 3:     iteration ← 0, non_improving ← 0, best ← ∞
 4:     k ← ∅, tabu ← ∅, history ← ∅
 5:     for all p ∈ P do                                  ▷ Initialise s.t. c1 is respected
 6:         Add a random mapping s_(h,p) ↦ G', where G' ⊂ G, for each h ∈ H to k
 7:         s.t. penalty(Partition({s_(h,p) | h ∈ H}, G))(k) = 0
 8:     while penalty(⟨V, D, C⟩)(k) > 0 & iteration < MaxIter do
 9:         iteration ← iteration + 1, non_improving ← non_improving + 1
10:         choose s_(h,p) ∈ V s.t. ∀s' ∈ V : conflict(s_(h,p), C)(k) ≥ conflict(s', C)(k)
11:         N ← move(s_(h,p), {s_(h',p) | h' ∈ H & h' ≠ h})(k)
12:         choose k' ∈ N s.t. ∀k'' ∈ N : penalty(⟨V, D, C⟩)(k') ≤
                penalty(⟨V, D, C⟩)(k'')
13:         and ((s_(h',p), d, iteration) ∉ tabu or penalty(⟨V, D, C⟩)(k') < best),
14:         where delta(k, k') = {(s_(h,p), {d}, ∅), (s_(h',p), ∅, {d})}
15:         k ← k', tabu ← tabu ∪ {(s_(h',p), d, iteration + rand_int(5, 40))}
16:         if penalty(⟨V, D, C⟩)(k) < best then
17:             best ← penalty(⟨V, D, C⟩)(k), non_improving ← 0,
18:             history ← {k}, tabu ← ∅
19:         else if penalty(⟨V, D, C⟩)(k) = best then
20:             history ← history ∪ {k}
21:         else if non_improving = MaxNonImproving then
22:             k ← a random element in history
23:             non_improving ← 0, history ← {k}, tabu ← ∅
24:     return k
```

The algorithm starts by initialising a CSP for the PPP, necessary counters, bounds, and sets (lines $2-4$), as well as the variables of the problem (lines $5-7$). As long as the penalty is positive and a maximum number of iterations has not been reached, lines $8-23$ explore the neighbourhood of the problem in the following way. (i) Choose a variable $s_{(h,p)}$ with maximum conflict (line 10). (ii) Determine the neighbourhood of type *move* for $s_{(h,p)}$ with respect to the other variables in the same period (line 11). (iii) Move to a neighbour k' that minimises the penalty (lines $12-14$).

In order to escape local minima it also uses a tabu list and a restarting component. The tabu list *tabu* is initially empty. When a move from a configuration k to a configuration k' is performed, meaning that for two variables $s_{(h,p)}$ and $s_{(h',p)}$,

a value d in $k(s_{(h,p)})$ is moved to $k(s_{(h',p)})$, the triple $(s_{(h'p)}, d, iteration + t)$ is added to *tabu*. This means that d cannot be moved to $s_{(h',p)}$ again for the next t iterations, where t is a random number between 5 and 40 (empirically chosen). However, if such a move would imply the lowest penalty so far, it is always accepted (lines $13-15$). By abuse of notation, we let $(s, d, t) \notin tabu$ be false iff $(s, d, t') \in tabu \ \ \& \ \ t \leq t'$.

The restarting component (lines $16-23$) works in the following way. Each configuration k such that $penalty(\langle V, D, C\rangle)(k)$ is at most the current lowest penalty is stored in the set *history* (lines $16-20$). If a number *MaxNonImproving* of iterations passes without any improvement to the lowest overall penalty, then the search is restarted from a random element in *history* (lines $21-23$). A similar restarting component was used in [13, 24] (saving one best configuration) and [6] (saving a set of best configurations), both for integer-domain models of the PPP.

5.2 The Social Golfers Problem (SGP)

In a golf club, there is a set of golfers, each of whom play golf once a week (constraint c_1) and always in ng groups of size ns (constraint c_2). The objective is to determine whether there is a schedule of nw weeks of play for these golfers, such that there is at most one week where any two distinct players are scheduled to play in the same group (constraint c_3).

A Set-Based Model. Let G be the set of golfers and let $s_{(g,w)}$ be a set variable containing the players playing in group g in week w. Then the following constraints model the problem:

$(c_1) : \forall w \in 1 \ldots nw : Partition(\{s_{(g,w)} \mid g \in 1 \ldots ng\}, G)$
$(c_2) : \forall g \in 1 \ldots ng : \forall w \in 1 \ldots nw : Cardinality(s_{(g,w)}, ns)$
$(c_3) : MaxIntersect(\{s_{(g,w)} \mid i \in 1 \ldots ng \ \ \& \ \ w \in 1 \ldots nw\}, 1)$

Solving The SGP. Similar to the PPP, we need to define an initial configuration and a neighbourhood for the SGP. This, and a slightly changed tabu list, are the only changes in the algorithm compared to the one we used for the PPP, hence the algorithm for the SGP is not shown.

We choose an initial configuration k and a neighbourhood that respects the constraints c_1 and c_2, i.e. that each golfer plays every week and that each group is of size ns. We do this (i) by assigning random disjoint subsets of size ns of G to each $s_{(g,w)}$ where $g \in 1 \ldots ng$ for each week $w \in 1 \ldots nw$ and (ii) by choosing the neighbourhood called *swap*, specifying the swap of two distinct golfers between a given group g and another group g' in the same week. Given such a swap of golfers between two different groups $s_{(g,w)}$ and $s_{(g',w)}$, what is now inserted in the tabu list are both $(s_{(g,w)}, d, t)$ and $(s_{(g',w)}, d, t)$ with t being as for the PPP.

5.3 Results

Tables 1 and 2 show the experimental results for the PPP and SGP, respectively. For both, each entry in the table is the mean value of successful runs out of 100.

Table 1. Run times in seconds for the PPP. Mean run time of successful runs (out of 100) and number of unsuccessful runs (if any) in parentheses

H/periods (fails)	6	7	8	9	10
1-12,16			1.2	2.3	21.0
1-13			7.0	90.5	
1,3-13,19			7.2	128.4 (4)	
3-13,25,26			13.9	170.0 (17)	
1-11,19,21	10.3	83.0 (1)			
1-9,16-19	18.2	160.6 (22)			

Table 2. Run times in seconds for the SGP. Mean run time of successful runs (out of 100) and number of unsuccessful runs (if any) in parentheses

ng-ns-nw	time	(fails)	*ng-ns-nw*	time	(fails)	*ng-ns-nw*	time	(fails)
6-3-7	0.4		6-3-8	215.0	(76)	7-3-9	138.0	(5)
8-3-10	14.4		9-3-11	3.5		10-3-13	325.0	(35)
6-4-5	0.3		6-4-6	237.0	(62)	7-4-7	333.0	(76)
8-4-7	0.9		8-4-8	290.0	(63)	9-4-8	1.7	
10-4-9	2.5		6-5-5	101.0	(1)	7-5-5	1.3	
8-5-6	8.6		9-5-6	0.9		10-5-7	1.7	
6-6-3	0.2		7-6-4	1.2		8-6-5	18.6	
9-6-5	1.0		10-6-6	3.7		7-7-3	0.3	
8-7-4	4.9		9-7-4	0.8		10-7-5	3.4	
8-8-3	0.5		9-8-3	0.6		10-8-4	1.4	
9-9-3	0.7		10-9-3	0.8		10-10-3	1.1	

The numbers in parentheses are the numbers of unsuccessful runs, if any, for that instance. We empirically chose $MaxIter = 500,000$ and $MaxNonImproving = 500$ for both applications. For the PPP, the instances are the same as in [25, 6, 24] and for the SGP, the instances are taken from [9]. For both applications, our results are comparable to, but not quite as fast as, the current best results ([6, 24] and [9] respectively) that we are aware of. We believe that they can be improved by using more sophisticated neighbourhoods and meta-heuristics, as well as by implementing the ideas in Section 3.3 for the *MaxIntersect* constraint.

6 Conclusion

We have proposed to use set variables and set constraints in local search. In order to do this, we have introduced a generic penalty scheme for (global) set constraints and used it to give incrementally maintainable penalty definitions for five such constraints. These were then used to model and solve two well-known combinatorial problems.

This research is motivated by the fact that set variables may lead to more intuitive and simpler problem models, providing the user with a richer set of tools, as well as more preserved structure in underlying solving algorithms such as the incremental algorithms for maintaining penalties: (global) set constraints provide opportunities for hard-wired global reasoning that would otherwise have to be hand-coded each time for lower-level encodings of set variables.

In terms of related work, Localizer [12, 22], by Michel and Van Hentenryck, was the first modelling language to allow the definition of local search algorithms in a high-level, declarative way. It introduces invariants to obtain efficient incremental algorithms. It also stresses the need for globality by making explicit the invariants *distribute* and *dcount*.

In [10], Galinier and Hao use a similar scheme to ours for defining the penalty of a constraint in local search: they define as the penalty of a (global) constraint c the minimum number of variables in c that must change in order for it to be satisfied. Note, however, that this work is for integer variables only. Nareyek uses global constraints in [16] and argues that this is a good compromise between low-level CSP approaches, using only simple (e.g., binary) constraints, and problem-tailored local search approaches that are hard to reuse.

Comet [13], also by Van Hentenryck and Michel, is an object-oriented language tailored for the elegant modelling and solving of combinatorial problems. With Comet, the concept of differentiable object was introduced, which is an abstraction that reconciles incremental computation and global constraints. A differentiable object may for instance be queried to evaluate the effect of local moves. Comet also introduced abstractions for controlling search [23] and modelling using constraint-based combinators such as logical operators and reification [24]. Both Localizer and Comet support set invariants, but these are not used as variables directly in constraints.

Generic penalty definitions for constraints are useful also in the soft-constraints area. Petit *et al.* [17] use a similar penalty definition to the one of Galinier and Hao [10] as well as another definition where the primal graph of a constraint is used to determine its cost. This definition of cost is then refined by Petit and Beldiceanu in [5], where the cost is expressed in terms of graph properties [4]. Bohlin [6] also introduces a scheme built on the graph properties in [4] for defining penalties, which is used in his Composer library for local search. To our knowledge, none of these approaches considers set variables and set constraints.

Open issues exist as well. Other than fine-tuning the performance of our current prototype implementation, further (global) set constraints should be added. What impact will a change to the penalty of *MaxIntersect* with respect to Section 3.3 have? In what way should the penalties of the (global) set constraints in this paper be generalised to allow problems containing variables with several kinds of domains? For instance, it would be useful to be able to replace m with an integer variable in the *Cardinality*, *MaxIntersect*, and *MaxWeightedSum* constraints, to allow negative weights in the latter, and to have a variable reference set in the *Partition* constraint.

Overall, our results are already very promising and motivate such further research.

Acknowledgements. This research was partially funded by Project C/1.246/HQ/JC/04 of EuroControl. We thank the referees for their useful comments.

References

1. E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons Ltd., 1997.
2. F. Azevedo and P. Barahona. Applications of an extended set constraint solver. In *Proc. of the ERCIM / CompulogNet Workshop on Constraints*, 2000.
3. N. Barnier and P. Brisset. Solving the Kirkman's schoolgirl problem in a few seconds. In *Proc. of CP'02*, volume 2470 of *LNCS*, pages 477–491. Springer, 2002.
4. N. Beldiceanu. Global constraints as graph properties on a structured network of elementary constraints of the same type. In *Proc. of CP'00*, volume 1894 of *LNCS*, pages 52–66. Springer-Verlag, 2000.
5. N. Beldiceanu and T. Petit. Cost evaluation of soft global constraints. In *Proc. of CPAIOR'04*, volume 3011 of *LNCS*, pages 80–95. Springer-Verlag, 2004.
6. M. Bohlin. Design and Implementation of a Graph-Based Constraint Model for Local Search. PhL thesis, Mälardalen University, Västerås, Sweden, April 2004.
7. P. Codognet and D. Diaz. Yet another local search method for constraint solving. In *Proc. of SAGA'01*, volume 2264 of *LNCS*, pages 73–90. Springer-Verlag, 2001.
8. K. Corrádi. Problem at Schweitzer competition. *Mat. Lapok*, 20:159–162, 1969.
9. I. Dotú and P. Van Hentenryck. Scheduling social golfers locally. In *Proc. of CPAIOR'05*, *LNCS*, Springer-Verlag, 2005.
10. P. Galinier and J.-K. Hao. A general approach for constraint solving by local search. In *Proc. of CP-AI-OR'00*.
11. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
12. L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. In *Proc. of CP'97*, volume 1330 of *LNCS*. Springer-Verlag, 1997.
13. L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):101–110, 2002. *Proc. of OOPSLA'02*.
14. L. Michel and P. Van Hentenryck. Maintaining longest paths incrementally. In *Proc. of CP'03*, volume 2833 of *LNCS*, pages 540–554. Springer-Verlag, 2003.
15. T. Müller and M. Müller. Finite set constraints in Oz. In *Proc. of 13th Workshop Logische Programmierung*, pages 104–115, Technische Universität München, 1997.
16. A. Nareyek. Using global constraints for local search. In *Constraint Programming and Large Scale Discrete Optimization*, volume 57 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 9–28. AMS, 2001.
17. T. Petit, J.-C. Régin, and C. Bessière. Specific filtering algorithms for over constrained problems. In *Proc. of CP'01*, volume 2293 of *LNCS*, Springer, 2001.
18. S. Prestwich. Supersymmetric modeling for local search. In *Proc. of 2nd International Workshop on Symmetry in Constraint Satisfaction Problems, at CP'02*.
19. J.-F. Puget. Finite set intervals. In *Proc. of CP'96 Workshop on Set Constraints*.
20. J.-F. Puget. Symmetry breaking revisited. In *Proc. of CP'02*, volume 2470 of *LNCS*, pages 446–461. Springer-Verlag, 2002.
21. B. M. Smith *et al.* The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1996.
22. P. Van Hentenryck and L. Michel. Localizer. *Constraints*, 5(1–2):43–84, 2000.
23. P. Van Hentenryck and L. Michel. Control abstractions for local search. In *Proc. of CP'03*, volume 2833 of *LNCS*, pages 65–80. Springer-Verlag, 2003.
24. P. Van Hentenryck, L. Michel, and L. Liu. Constraint-based combinators for local search. In *Proc. of CP'04*, volume 3258 of *LNCS*. Springer-Verlag, 2004.
25. J. P. Walser. *Integer Optimization by Local Search: A Domain-Independent Approach*, volume 1637 of *LNCS*. Springer-Verlag, 1999.

Paper C

Incremental Algorithms for Local Search from Existential Second-Order Logic

To be published in:

P. van Beek (editor), Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming - CP 2005, Lecture Notes in Computer Science, Springer-Verlag, 2005.

Incremental Algorithms for Local Search from Existential Second-Order Logic

Magnus Ågren, Pierre Flener, and Justin Pearson

Department of Information Technology
Uppsala University, Box 337, SE – 751 05 Uppsala, Sweden
{agren,pierref,justin}@it.uu.se

Abstract. Local search is a powerful and well-established method for solving hard combinatorial problems. Yet, until recently, it has provided very little user support, leading to time-consuming and error-prone implementation tasks. We introduce a scheme that, from a high-level description of a constraint in existential second-order logic with counting, automatically synthesises incremental penalty calculation algorithms. The performance of the scheme is demonstrated by solving real-life instances of a financial portfolio design problem that seem unsolvable in reasonable time by complete search.

1 Introduction

Local search is a powerful and well-established method for solving hard combinatorial problems [1]. Yet, until recently, it has provided very little user support, leading to time-consuming and error-prone implementation tasks. The recent emergence of languages and systems for local search, sometimes based on novel abstractions, has alleviated the user of much of this burden [10, 16, 12, 11].

However, if a problem cannot readily be modelled using the primitive constraints of such a local search system, then the *user* has to perform some of those time-consuming and error-prone tasks. These include the design of algorithms for the calculation of penalties of user-defined constraints. These algorithms are called very often in the innermost loop of local search and thus need to be implemented particularly efficiently: incrementality is crucial. Would it thus not be nice if also this task could be performed fully automatically and satisfactorily by a local search system? In this paper, we design a scheme for doing just that, based on an extension of the idea of combinators [15] to quantifiers. Our *key contributions* are as follows:

- We propose the usage of existential second-order logic with counting as a *high-level modelling language* for (user-defined) constraints. It accommodates set variables and captures at least the complexity class NP.
- We design a scheme for the *automated synthesis of incremental penalty calculation algorithms* from a description of a (user-defined) constraint in that language. We have developed an *implementation* of this scheme.

– We propose a *new benchmark problem* for local search, with applications in finance. Using our local search framework, we *exactly solve real-life instances* that seem unsolvable in reasonable time by complete search; the performance is competitive with a fast approximation method based on complete search.

The rest of this paper is organised as follows. In Section 2, we define the background for this work, namely constraint satisfaction problems over scalar and set variables as well as local search concepts. The core of this paper are Sections 3 to 6, where we introduce the used modelling language and show how incremental algorithms for calculating penalties can be automatically synthesised from a model therein. In Section 7, we demonstrate the performance of this approach by solving real-life instances of a financial portfolio design problem. Finally, we summarise our results, discuss related work, and outline future work in Section 8.

2 Preliminaries

As usual, a *constraint satisfaction problem* (*CSP*) is a triple $\langle V, D, C\rangle$, where V is a finite set of variables, D is a finite set of domains, each $D_v \in D$ containing the set of possible values for the corresponding variable $v \in V$, and C is a finite set of constraints, each $c \in C$ being defined on a subset of the variables in V and specifying their valid combinations of values.

Definition 1 (Set Variable and its Universe). *Let $P = \langle V, D, C\rangle$ be a CSP. A variable $S \in V$ is a* set variable *if its corresponding domain $D_S = 2^{\mathcal{U}_S}$, where $\mathcal{U}_S$ is a finite set of values of some type, called the* universe *of S.*

Without loss of generality, we assume that all the set variables have a common universe, denoted $\mathcal{U}$. We also assume that *all* the variables are set variables, and denote such a set-CSP by $\langle V, \mathcal{U}, C\rangle$. This is of course a limitation, since many models contain both set variables and scalar variables. Fortunately, interesting applications, such as the ones in this paper and in [2], can be modelled using only set variables.

A constraint program assigns values to the variables one by one, but local search maintains an (initially arbitrary) assignment of values to *all* the variables:

Definition 2 (Configuration). *Let $P = \langle V, \mathcal{U}, C\rangle$ be a set-CSP. A* configuration *for P (or V) is a total function $k : V \to 2^{\mathcal{U}}$.*

As usual, the notation $k \models \phi$ expresses that the open formula ϕ is satisfied under the configuration k.

Example 1. Consider a set-CSP $P = \langle\{S_1, S_2, S_3\}, \{d_1, d_2, d_3\}, \{c_1, c_2\}\rangle$. A configuration for P is given by $k(S_1) = \{d_3\}, k(S_2) = \{d_1, d_2\}, k(S_3) = \emptyset$, or equivalently as the set of mappings $\{S_1 \mapsto \{d_3\}, S_2 \mapsto \{d_1, d_2\}, S_3 \mapsto \emptyset\}$. Another configuration for P is given by $k' = \{S_1 \mapsto \emptyset, S_2 \mapsto \{d_1, d_2, d_3\}, S_3 \mapsto \emptyset\}$.

Local search iteratively makes a small change to the current configuration, upon examining the merits of many such changes. The configurations thus examined constitute the neighbourhood of the current configuration:

Definition 3 (Neighbourhood). *Let K be the set of all configurations for a (set-)CSP P and let $k \in K$. A* neighbourhood function *for P is a function $\mathcal{N} : K \to 2^K$. The* neighbourhood *of P with respect to k and $\mathcal{N}$ is the set $\mathcal{N}(k)$.*

Example 2. Reconsider P and k from Example 1. A neighbourhood of P with respect to k and some neighbourhood function for P is the set $\{k_1 = \{S_1 \mapsto \emptyset, S_2 \mapsto \{d_1, d_2, d_3\}, S_3 \mapsto \emptyset\}, k_2 = \{S_1 \mapsto \emptyset, S_2 \mapsto \{d_1, d_2\}, S_3 \mapsto \{d_3\}\}$. This neighbourhood function moves the value d_3 in S_1 to S_2 or S_3.

The penalty of a CSP is an estimate on how much its constraints are violated:

Definition 4 (Penalty). *Let $P = \langle V, D, C \rangle$ be a (set-)CSP and let K be the set of all configurations for P. A* penalty function *of a constraint $c \in C$ is a function $penalty(c) : K \to \mathbb{N}$ such that $penalty(c)(k) = 0$ if and only if c is satisfied under configuration k. The* penalty *of a constraint $c \in C$ with respect to a configuration $k \in K$ is $penalty(c)(k)$. The* penalty *of P with respect to a configuration $k \in K$ is the sum $\sum_{c \in C} penalty(c)(k)$.*

Example 3. Consider once again P from Example 1 and let c_1 and c_2 be the constraints $S_1 \subseteq S_2$ and $d_3 \in S_3$ respectively. Let the penalty functions of c_1 and c_2 be defined by $penalty(c_1)(k) = |k(S_1) \setminus k(S_2)|$ and $penalty(c_2)(k) = 0$ if $d_3 \in k(S_3)$ and 1 otherwise. Now, the penalties of P with respect to the configurations k_1 and k_2 from Example 2 are $penalty(c_1)(k_1) + penalty(c_2)(k_1) = 1$ and $penalty(c_1)(k_2) + penalty(c_2)(k_2) = 0$, respectively.

3 Second-Order Logic

We use *existential second-order logic* ($\exists$SOL) [8], extended with counting, for modelling the constraints of a set-CSP. $\exists$SOL is very expressive: it captures the complexity class NP [5]. Figure 1 shows the BNF grammar for the used language, which we will refer to as $\exists\text{SOL}^+$. Some of the production rules are highlighted and the reason for this is explained below. The language uses common mathematical and logical notations. Note that its set of relational operators is closed under negation. A formula in $\exists\text{SOL}^+$ is of the form $\exists S_1 \cdots \exists S_n \phi$, i.e., a sequence of existentially quantified set variables, ranging over the power set of an implicit common universe $\mathcal{U}$, and constrained by a logical formula ϕ. The usual precedence rules apply when parentheses are omitted, i.e., $\neg$ has highest precedence, $\wedge$ has higher precedence than $\vee$, etc.

Example 4. The constraint $S \subset T$ on the set variables S and T may be expressed in $\exists\text{SOL}^+$ by the formula:

$$\exists S \exists T((\forall x(x \notin S \vee x \in T)) \wedge (\exists x(x \in T \wedge x \notin S))) \tag{1}$$

The constraint $|S \cap T| \leq m$ on the set variables S and T and the natural-number constant m may be expressed in $\exists\text{SOL}^+$ by the formula:

$$\exists S \exists T \exists I((\forall x(x \in I \leftrightarrow x \in S \wedge x \in T)) \wedge |I| \leq m) \tag{2}$$

Note that we used an additional set variable I to represent the intersection $S \cap T$.

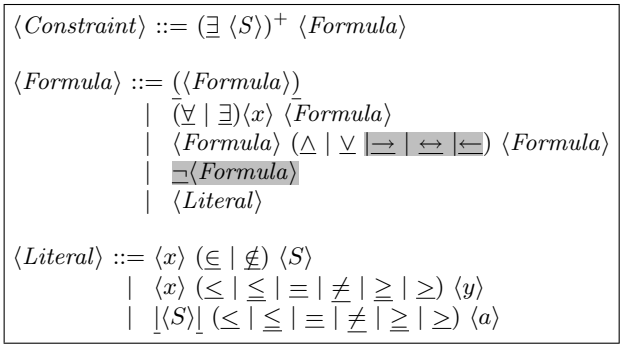


Fig. 1. The BNF grammar for the language $\exists\text{SOL}^+$ where terminal symbols are underlined. The non-terminal symbol $\langle S\rangle$ denotes an identifier for a bound set variable S such that $S \subseteq \mathcal{U}$, while $\langle x\rangle$ and $\langle y\rangle$ denote identifiers for bound variables x and y such that $x, y \in \mathcal{U}$, and $\langle a\rangle$ denotes a natural number constant. The core subset of $\exists\text{SOL}^+$ corresponds to the language given by the non-highlighted production rules.

In Section 4 we will define the penalty of formulas in $\exists\text{SOL}^+$. Before we do this, we define a core subset of this language that will be used in that definition. This is only due to the way we define the penalty and does not pose any limitations on the expressiveness of the language: Any formula in $\exists\text{SOL}^+$ may be transformed into a formula in that core subset, in a way shown next.

The transformations are standard and are only described briefly. First, given a formula $\exists S_1 \cdots \exists S_n \phi$ in $\exists\text{SOL}^+$, we remove its negations by pushing them downward, all the way to the literals of ϕ, which are replaced by their negated counterparts. Assuming that ϕ is the formula $\forall x(\neg(x \in S \wedge x \notin S'))$, it is transformed into $\forall x(x \notin S \vee x \in S')$. This is possible because the set of relational operators in $\exists\text{SOL}^+$ is closed under negation. Second, equivalences are transformed into conjunctions of implications, which are in turn transformed into disjunctions. Assuming that ϕ is the formula $\forall x(x \in S_1 \leftrightarrow x \in S_2)$, it is transformed into $\forall x((x \notin S_1 \vee x \in S_2) \wedge (x \in S_1 \vee x \notin S_2))$.

By performing these transformations for ϕ (and recursively for the subformulas of ϕ) in any formula $\exists S_1 \cdots \exists S_n \phi$, we end up with the non-highlighted subset of the language in Figure 1, for which we will define the penalty.

Example 5. (1) is in the core subset of $\exists\text{SOL}^+$. The core equivalent of (2) is:

$$\exists S \exists T \exists I((\forall x((x \notin I \vee x \in S \wedge x \in T) \wedge (x \in I \vee x \notin S \vee x \notin T))) \wedge |I| \leq m) \quad (3)$$

From now on we assume that any formula said in $\exists\text{SOL}^+$ is already in the core subset of $\exists\text{SOL}^+$. The full language just offers convenient shorthand notations.

4 The Penalty of an $\exists\text{SOL}^+$ Formula

In order to use (closed) formulas in $\exists\text{SOL}^+$ as constraints in our local search framework, we must define the penalty function of such a formula according

to Definition 4, which is done inductively below. It is important to stress that *this calculation is totally generic and automatable, as it is based only on the syntax of the formula and the semantics of the quantifiers, connectives, and relational operators of the* $\exists\text{SOL}^+$ *language, but not on the intended semantics of the formula. A human might well give a different penalty function to that formula, and a way of calculating it that better exploits globality, but the scheme below requires* no *such user participation.*

We need to express the penalty with respect to the values of any bound first-order variables. We will therefore pass around an (initially empty) environment Γ in the definition below, where Γ is a total function from the currently bound first-order variables into the common universe of values.

Definition 5 (Penalty of an $\exists\text{SOL}^+$ Formula). *Let $\mathcal{F}$ be a formula in $\exists\text{SOL}^+$ of the form $\exists S_1 \cdots \exists S_n \phi$, let k be a configuration for $\{S_1, \ldots, S_n\}$, and let Γ be an environment. The* penalty *of $\mathcal{F}$ with respect to k and Γ is given by a function penalty′ defined by:*

$$
\begin{aligned}
&(a)\ \mathit{penalty}'(\Gamma)(\exists S_1 \cdots \exists S_n \phi)(k) &&= \mathit{penalty}'(\Gamma)(\phi)(k)\\
&(b)\ \mathit{penalty}'(\Gamma)(\forall x \phi)(k) &&= \sum_{u \in \mathcal{U}} \mathit{penalty}'(\Gamma \cup \{x \mapsto u\})(\phi)(k)\\
&(c)\ \mathit{penalty}'(\Gamma)(\exists x \phi)(k) &&= \min\{\mathit{penalty}'(\Gamma \cup \{x \mapsto u\})(\phi)(k) \mid u \in \mathcal{U}\}\\
&(d)\ \mathit{penalty}'(\Gamma)(\phi \wedge \psi)(k) &&= \mathit{penalty}'(\Gamma)(\phi)(k) + \mathit{penalty}'(\Gamma)(\psi)(k)\\
&(e)\ \mathit{penalty}'(\Gamma)(\phi \vee \psi)(k) &&= \min\{\mathit{penalty}'(\Gamma)(\phi)(k), \mathit{penalty}'(\Gamma)(\psi)(k)\}\\
&(f)\ \mathit{penalty}'(\Gamma)(x \leq y)(k) &&= \begin{cases} 0, \text{ if } \Gamma(x) \leq \Gamma(y) \\ 1, \text{ otherwise} \end{cases}\\
&(g)\ \mathit{penalty}'(\Gamma)(|S| \leq c)(k) &&= \begin{cases} 0, \text{ if } |k(S)| \leq c \\ |k(S)| - c, \text{ otherwise} \end{cases}\\
&(h)\ \mathit{penalty}'(\Gamma)(x \in S)(k) &&= \begin{cases} 0, \text{ if } \Gamma(x) \in k(S) \\ 1, \text{ otherwise} \end{cases}\\
&(i)\ \mathit{penalty}'(\Gamma)(x \notin S)(k) &&= \begin{cases} 0, \text{ if } \Gamma(x) \notin k(S) \\ 1, \text{ otherwise} \end{cases}
\end{aligned}
$$

Now, the penalty function *of $\mathcal{F}$ is the function $\mathit{penalty}(\mathcal{F}) = \mathit{penalty}'(\emptyset)(\mathcal{F})$.*

In the definition above, for (sub)formulas of the form $x \Diamond y$ and $|S| \Diamond c$, where $\Diamond \in \{<, \leq, =, \neq, \geq, >\}$, we only show the cases where $\Diamond \in \{\leq\}$; the other cases are defined similarly. (The same applies to the algorithms in Section 5.) The following proposition is a direct consequence of the definition above:

Proposition 1. *The penalty of a formula $\mathcal{F}$ with respect to a configuration k is 0 if and only if $\mathcal{F}$ is satisfied under k: $\mathit{penalty}(\exists S_1 \cdots \exists S_n \phi)(k) = 0 \Leftrightarrow k \models \phi$.*

In our experience, the calculated penalties of violated constraints are often meaningful, as shown in the following example.

Example 6. Let $\mathcal{U} = \{a, b\}$ and let k be the configuration for $\{S, T\}$ such that $k(S) = k(T) = \{a\}$. Let us calculate $\mathit{penalty}(\exists S \exists T \phi)(k)$, where $\exists S \exists T \phi$ is

the formula (1) The initial call matches case (a) which gives the recursive call $\mathit{penalty}'(\emptyset)(\phi)(k)$. Since ϕ is of the form $\psi \wedge \psi'$ this call matches case (d), which is defined as the sum of the recursive calls on ψ and ψ'. For the first recursive call, ψ is the formula $\forall x(x \notin S \vee x \in T)$. Hence it will match case (b), which is defined as the sum of the recursive calls $\mathit{penalty}'(\{x \mapsto a\})(x \notin S \vee x \in T)(k)$ and $\mathit{penalty}'(\{x \mapsto b\})(x \notin S \vee x \in T)(k)$ (one for each of the values a and b in $\mathcal{U}$). Both of these match case (e) which, for the first one, gives the minimum of the recursive calls $\mathit{penalty}'(\{x \mapsto a\})(x \notin S)(k)$ and $\mathit{penalty}'(\{x \mapsto a\})(x \in T)(k)$. This value is $\min\{1, 0\} = 0$ since $a \in T$. A similar reasoning for the second one gives the value $\min\{0, 1\} = 0$ as well since $b \notin S$. Hence the recursive call on ψ gives $0 + 0 = 0$. This means that ψ is satisfied and should indeed contribute nothing to the overall penalty. A similar reasoning for the recursive call on ψ', which is $\exists x(x \in T \wedge x \notin S)$, gives $\min\{1, 1\} = 1$. This means that ψ' is violated: the calculated contribution of 1 to the overall penalty means that no value of $\mathcal{U}$ belongs to T but not to S. Hence the returned overall penalty is $0 + 1 = 1$.

5 Incremental Penalty Maintenance using Penalty Trees

In our local search framework, given a formula $\mathcal{F}$ in $\exists\mathrm{SOL}^+$, we could use Definition 5 to calculate the penalty of $\mathcal{F}$ with respect to a configuration k, and then similarly for each configuration k' in a neighbourhood $\mathcal{N}(k)$ to be evaluated. However, a complete recalculation of the penalty with respect to Definition 5 is impractical, since $\mathcal{N}(k)$ is usually a very large set.

In local search it is crucial to use *incremental algorithms* when evaluating the penalty of a constraint with respect to a neighbour k' to a current configuration k. We will now present a scheme for incremental maintenance of the penalty of a formula in $\exists\mathrm{SOL}^+$ with respect to Definition 5. This scheme is based on viewing a formula $\mathcal{F}$ in $\exists\mathrm{SOL}^+$ as a syntax tree and observing that, given the penalty with respect to k, only the paths from the leaves that contain variables that are changed in k' compared to k to the root node need to be updated to obtain the penalty with respect to k'.

5.1 The Penalty Tree of a Formula

First, a syntax tree $\mathbf{T}$ of a formula $\mathcal{F}$ in $\exists\mathrm{SOL}^+$ of the form $\exists S_1 \cdots \exists S_n \phi$ is constructed in the usual way. Literals in $\mathcal{F}$ of the form $x \in S$, $x \notin S$, $x \Diamond y$, and $|S| \Diamond k$ (where $\Diamond \in \{<, \leq, =, \neq, \geq, >\}$) are leaves in $\mathbf{T}$. Subformulas in $\mathcal{F}$ of the form $\psi \Box \psi'$ (where $\Box \in \{\wedge, \vee\}$) are subtrees in $\mathbf{T}$ with $\Box$ as parent node and the trees of ψ and ψ' as children. When possible, formulas of the form $\psi_1 \Box \cdots \Box \psi_m$ give rise to one parent node with m children. Subformulas in $\mathcal{F}$ of the form $\forall x \psi$ (resp. $\exists x \psi$) are subtrees in $\mathbf{T}$ with $\forall x$ (resp. $\exists x$) as parent node and the tree of ψ as only child. Finally, $\exists S_1 \cdots \exists S_n$ is the root node of $\mathbf{T}$ with the tree of ϕ as child. As an example of this, Figure 2 shows the syntax tree of formula (3). Note that it contains additional information, to be explained in Section 5.2.

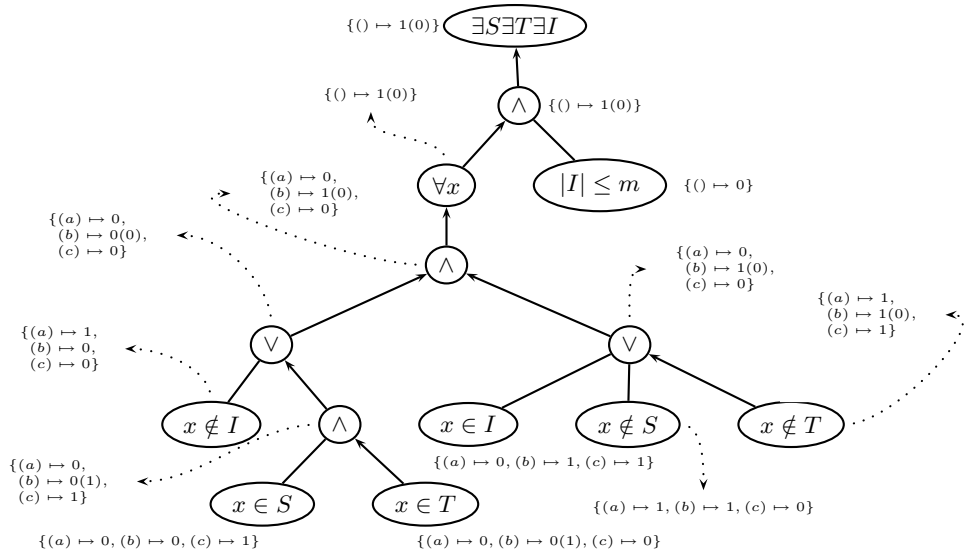


Fig. 2. Penalty tree of formula (3).

Assume that $\mathbf{T}$ is the syntax tree of a formula $\mathcal{F} = \exists S_1 \cdots \exists S_n \phi$. We will now extend $\mathbf{T}$ into a penalty tree in order to obtain incremental penalty maintenance of $\mathcal{F}$. Given an initial configuration k for $\{S_1, \ldots, S_n\}$, the penalty with respect to k of the subformula that the tree rooted at node $\mathbf{n}$ represents is stored in each node $\mathbf{n}$ of $\mathbf{T}$. This implies that the penalty stored in the root node of $\mathbf{T}$ is equal to $\mathit{penalty}(\mathcal{F})(k)$. When a configuration k' in the neighbourhood of k is to be evaluated, the only paths in $\mathbf{T}$ that may have changed are those leading from leaves containing any of the set variables S_i that are affected by the change of k to k'. By starting at each of these leaves $l(S_i)$ and updating the penalty with respect to the change of S_i of each node on the path from l to the root node of $\mathbf{T}$, we can incrementally calculate $\mathit{penalty}(\mathcal{F})(k')$ given k.

5.2 Initialising the Nodes with Penalties

For the descendants of nodes representing subformulas that introduce bound variables, we must store the penalty with respect to *every* possible mapping of those variables. For example, the child node $\mathbf{n}$ of a node for a subformula of the form $\forall x \phi$ will have a penalty stored for each $u \in \mathcal{U}$. Generally, the penalty stored at a node $\mathbf{n}$ is a mapping, denoted $p(\mathbf{n})$, from the possible tuples of values of the bound variables at $\mathbf{n}$ to $\mathbb{N}$. Assume, for example, that at $\mathbf{n}$ there are two bound variables x and y (introduced in that order) and that $\mathcal{U} = \{a, b\}$. Then the penalty stored at $\mathbf{n}$ after initialisation will be the mapping $\{(a,a) \mapsto p_1, (a,b) \mapsto p_2, (b,a) \mapsto p_3, (b,b) \mapsto p_4\}$ where $\{p_1, p_2, p_3, p_4\} \subset \mathbb{N}$. The first element of each tuple corresponds to x and the second one to y. If there are no bound variables at a particular node, then the penalty is a mapping $\{() \mapsto q\}$, i.e., the empty tuple mapped to some $q \in \mathbb{N}$.

Algorithm 1 Initialises the penalty mappings of a penalty tree.

function $initialise(\mathbf{T}, \Gamma, \mathcal{U}, k)$
  match T with
    $\exists S_1 \cdots \exists S_n \phi \longrightarrow p(\mathbf{T}) \leftarrow \{tuple(\Gamma) \mapsto initialise(\phi, \Gamma, \mathcal{U}, k)\}$
  | $\forall x \phi \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \{tuple(\Gamma) \mapsto \sum_{u \in \mathcal{U}} initialise(\phi, \Gamma \cup \{x \mapsto u\}, \mathcal{U}, k)\}$
  | $\exists x \phi \longrightarrow$
      $p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \{tuple(\Gamma) \mapsto \min\{initialise(\phi, \Gamma \cup \{x \mapsto u\}, \mathcal{U}, k) \mid u \in \mathcal{U}\}\}$
  | $\phi_1 \wedge \cdots \wedge \phi_m \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \{tuple(\Gamma) \mapsto \sum_{1 \leq i \leq m} initialise(\phi_i, \Gamma, \mathcal{U}, k)\}$
  | $\phi_1 \vee \cdots \vee \phi_m \longrightarrow$
      $p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \{tuple(\Gamma) \mapsto \min\{initialise(\phi, \Gamma, \mathcal{U}, k) \mid \phi \in \{\phi_1, \ldots, \phi_m\}\}\}$

  $$\mid x \leq y \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \left\{ tuple(\Gamma) \mapsto \begin{cases} 0, & \text{if } \Gamma(x) \leq \Gamma(y) \\ 1, & \text{otherwise} \end{cases} \right\}$$

  $$\mid |S| \leq m \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \left\{ tuple(\Gamma) \mapsto \begin{cases} 0, & \text{if } |k(S)| \leq m \\ |k(S)| - m, & \text{otherwise} \end{cases} \right\}$$

  $$\mid x \in S \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \left\{ tuple(\Gamma) \mapsto \begin{cases} 0, & \text{if } \Gamma(x) \in k(S) \\ 1, & \text{otherwise} \end{cases} \right\}$$

  $$\mid x \notin S \longrightarrow p(\mathbf{T}) \leftarrow p(\mathbf{T}) \cup \left\{ tuple(\Gamma) \mapsto \begin{cases} 0, & \text{if } \Gamma(x) \notin k(S) \\ 1, & \text{otherwise} \end{cases} \right\}$$

  end match
  return $p(\mathbf{T})(tuple(\Gamma))$
function $tuple(\Gamma)$
  return $(\Gamma(x_1), \ldots, \Gamma(x_n))$ ▷ $\{x_1, \ldots, x_n\} = domain(\Gamma)$, introduced into Γ in that order.

Algorithm 1 shows the function $initialise(\mathbf{T}, \Gamma, \mathcal{U}, k)$ that initialises a penalty tree $\mathbf{T}$ of a formula with penalty mappings with respect to an (initially empty) environment Γ, a universe $\mathcal{U}$, and a configuration k. By abuse of notation, we let formulas in $\exists\text{SOL}^+$ denote their corresponding penalty trees, e.g., $\forall x \phi$ denotes the penalty tree with $\forall x$ as root node and the tree representing ϕ as only child, $\phi_1 \wedge \cdots \wedge \phi_m$ denotes the penalty tree with $\wedge$ as root node and the subtrees of all the ϕ_i as children, etc. Note that we use an auxiliary function *tuple* that, given an environment Γ, returns the tuple of values with respect to Γ. We also assume that before *initialise* is called for a penalty tree $\mathbf{T}$, the penalty mapping of each node in $\mathbf{T}$ is the empty set.

Example 7. Let $k = \{S \mapsto \{a, b\}, T \mapsto \{a, b, c\}, I \mapsto \{a\}\}$, let $\mathcal{U} = \{a, b, c\}$, and let $m = 1$. Figure 2 shows the penalty tree $\mathbf{T}$ with penalty mappings (dotted arrows connect nodes to their mappings) after $initialise(\mathbf{T}, \emptyset, \mathcal{U}, k)$ has been called for formula (3). As can be seen at the root node, the initial penalty is 1. Indeed, there is *one* value, namely b, that is in S and T but not in I.

5.3 Maintaining the Penalties

We will now present a way of incrementally updating the penalty mappings of a penalty tree. This is based on the observation that, given an initialised penalty tree $\mathbf{T}$, a current configuration k, and a configuration to evaluate k', only the paths leading from any leaf in $\mathbf{T}$ affected by changing k to k' to the root node of $\mathbf{T}$ need to be updated.

Algorithm 2 shows the function $submit(\mathbf{n}, \mathbf{n}', \mathcal{A}, k, k')$ that updates the penalty mappings of a penalty tree incrementally. It is a recursive function where infor-

Algorithm 2 Updates the penalty mappings of a penalty tree.

function $submit(\mathbf{n}, \mathbf{n}', \mathcal{A}, k, k')$
  $update(\mathbf{n}, \mathbf{n}', \mathcal{A})$ ▷ First update $\mathbf{n}$ with respect to $\mathbf{n}'$.
  if All children affected by the change of k to k' are done **then**
    if $\mathbf{n}$ is not the root node **then**
      $submit(parent(\mathbf{n}), \mathbf{n}, \mathcal{A} \cup changed(\mathbf{n}), k, k')$
      $changed(\mathbf{n}) \leftarrow \emptyset$
    else () ▷ We are at the root. Done!
  else $changed(\mathbf{n}) \leftarrow changed(\mathbf{n}) \cup \mathcal{A}$ ▷ Not all children done. Save tuples and wait.
function $update(\mathbf{n}, \mathbf{n}', \mathcal{A})$
  $p'(\mathbf{n}) \leftarrow p(\mathbf{n})$ ▷ Save the old penalty mapping.
  for all $t \in \mathcal{A}|_{bounds(\mathbf{n})}$ **do**
    match n **with**
      $\exists S_1 \cdots \exists S_n \phi \longrightarrow p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \{() \mapsto p(\mathbf{n}')(())\}$
    | $\forall x \phi \longrightarrow$
        for all $t' \in \mathcal{A}|_{bounds(\mathbf{n}')}$ s.t. $t'|_{bounds(\mathbf{n})} = t$ **do**
          $p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \{t \mapsto p(\mathbf{n})(t) + p(\mathbf{n}')(t') - p'(\mathbf{n}')(t')\}$
    | $\exists x \phi \longrightarrow$
        for all $t' \in \mathcal{A}|_{bounds(\mathbf{n}')}$ s.t. $t'|_{bounds(\mathbf{n})} = t$ **do**
          Replace the value for t' in $min\_heap(\mathbf{n}, t)$ with $p(\mathbf{n}')(t')$
          $p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \{t \mapsto \min(min\_heap(\mathbf{n}, t))\}$
    | $\phi_1 \wedge \cdots \wedge \phi_m \longrightarrow p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \{t \mapsto p(\mathbf{n})(t) + p(\mathbf{n}')(t) - p'(\mathbf{n}')(t)\}$
    | $\phi_1 \vee \cdots \vee \phi_m \longrightarrow$ Replace the value for $\mathbf{n}'$ in $min\_heap(\mathbf{n}, t)$ with $p(\mathbf{n}')(t)$
        $p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \{t \mapsto \min(min\_heap(\mathbf{n}, t))\}$
    | $x \leq y \longrightarrow error$ ▷ Only leaves representing formulas on set variables apply!

    | $|S| \leq m \longrightarrow p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \left\{ t \mapsto \begin{cases} 0, \text{ if } |k'(S)| \leq m \\ |k'(S)| - m, \text{ otherwise} \end{cases} \right\}$

    | $x \in S \longrightarrow p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \left\{ t \mapsto \begin{cases} 0, \text{ if } t(x) \in k'(S) \\ 1, \text{ otherwise} \end{cases} \right\}$

    | $x \notin S \longrightarrow p(\mathbf{n}) \leftarrow p(\mathbf{n}) \oplus \left\{ t \mapsto \begin{cases} 0, \text{ if } t(x) \notin k'(S) \\ 1, \text{ otherwise} \end{cases} \right\}$
    end match

mation from the node $\mathbf{n}'$ (*void* when $\mathbf{n}$ is a leaf) is propagated to the node $\mathbf{n}$. The additional arguments are $\mathcal{A}$ (a set of tuples of values that are affected by changing k to k' at $\mathbf{n}$), k (the current configuration), and k' (the configuration to evaluate). It uses the auxiliary function $update(\mathbf{n}, \mathbf{n}', \mathcal{A})$ that performs the actual update of the penalty mappings of $\mathbf{n}$ with respect to (the change of the penalty mappings of) $\mathbf{n}'$.

The set $\mathcal{A}$ depends on the maximum number of bound variables in the penalty tree, the universe $\mathcal{U}$, and the configurations k and k'. Recall $\mathcal{U}$ and k of Example 7 and assume that $k' = \{S \mapsto \{a, b\}, T \mapsto \{a, c\}, I \mapsto \{a\}\}$ (b was removed from $k(T)$). In this case $\mathcal{A}$ would be the singleton set $\{(b)\}$ since this is the only tuple affected by the change of k to k'. However, if the maximum number of bound variables was two (instead of one as in Example 7), $\mathcal{A}$ would be the set $\{(b, a), (b, b), (b, c), (a, b), (c, b)\}$ since all of these tuples might be affected.

Some of the notation used in Algorithm 2 needs explanation: Given a set $\mathcal{A}$ of tuples, each of arity n, we use $\mathcal{A}|_m$ to denote the set of tuples in $\mathcal{A}$ projected on their first $m \leq n$ positions. For example, if $\mathcal{A} = \{(a, a), (a, b), (a, c), (b, a), (c, a)\}$, then $\mathcal{A}|_1 = \{(a), (b), (c)\}$. We use a similar notation for projecting a particular tuple: if $t = (a, b, c)$ then $t|_2$ denotes the tuple (a, b). We also use $t(x)$ to denote the value of the position of x in t. For example, if x was the second introduced

bound variable, then $t(x) = b$ for $t = (a, b, c)$. We let *changed*($\mathbf{n}$) denote the set of tuples that has affected $\mathbf{n}$. We let *bounds*($\mathbf{n}$) denote the number of bound variables at node $\mathbf{n}$ (which is equal to the number of nodes of the form $\forall x$ or $\exists x$ on the path from $\mathbf{n}$ to the root node). We use the operator $\oplus$ for replacing the current bindings of a mapping with new ones. For example, the result of $\{x \mapsto a, y \mapsto a, z \mapsto b\} \oplus \{x \mapsto b, y \mapsto b\}$ is $\{x \mapsto b, y \mapsto b, z \mapsto b\}$. Finally, we assume that nodes of the form $\exists x$ and $\vee$ have a data structure *min_heap* for maintaining the minimum value of each of its penalty mappings.

Now, given a change to a current configuration k, resulting in k', assume that $\{S_i\}$ is the set of affected set variables in a formula $\mathcal{F}$ with an initialised penalty tree $\mathbf{T}$. The call *submit*($\mathbf{n}$, *void*, $\mathcal{A}, k, k'$) must now be made for each leaf $\mathbf{n}$ of $\mathbf{T}$ that represents a subformula stated on S_i, where $\mathcal{A}$ is the set of affected tuples.

Example 8. Recall $k = \{S \mapsto \{a, b\}, T \mapsto \{a, b, c\}, I \mapsto \{a\}\}$ and $m = 1$ of Example 7, and keep the initialised tree $\mathbf{T}$ in Figure 2 in mind. Let $k' = \{S \mapsto \{a, b\}, T \mapsto \{a, c\}, I \mapsto \{a\}\}$, i.e., b was removed from $k(T)$. The function *submit* will now be called twice, once for each leaf in $\mathbf{T}$ containing T.

Starting with the leaf $\mathbf{n}_{11}$ representing the formula $x \in T$, *submit* is called with *submit*($\mathbf{n}_{11}$, *void*, $\{(b)\}, k, k'$). This gives the call *update*($\mathbf{n}_{11}$, *void*, $\{(b)\}$) which replaces the binding of (b) in $p(\mathbf{n}_{11})$ with $(b) \mapsto 1$ (since b is no longer in T). Since a leaf node has no children and $\mathbf{n}_{11}$ is not the root node, *submit*($\mathbf{n}_{12}$, $\mathbf{n}_{11}$, $\{(b)\}$, k, k') is called where $\mathbf{n}_{12} = \mathit{parent}(\mathbf{n}_{11})$. Since $\mathbf{n}_{12}$ is an $\wedge$-node, *update*($\mathbf{n}_{12}, \mathbf{n}_{11}, \{b\}$) implies that the binding of (b) in $p(\mathbf{n}_{12})$ is updated with the difference $p(\mathbf{n}_{11}) - p'(\mathbf{n}_{11})$ (which is 1 in this case). Hence, the new value of $p(\mathbf{n}_{12})(b)$ is 1. Since there are no other affected children of $\mathbf{n}_{12}$ and $\mathbf{n}_{12}$ is not the root node, *submit*($\mathbf{n}_{13}, \mathbf{n}_{12}, \{(b)\}, k, k'$) is called where $\mathbf{n}_{13} = \mathit{parent}(\mathbf{n}_{12})$. Since $\mathbf{n}_{13}$ is an $\vee$-node, *update*($\mathbf{n}_{13}, \mathbf{n}_{12}, \{b\}$) gives that the binding of (b) in $p(\mathbf{n}_{13})$ is updated with the minimum of $p(\mathbf{n}_{12})(b)$ and the values of $p(\mathbf{n})(b)$ for any other child $\mathbf{n}$ of $\mathbf{n}_{13}$. Since the only other child of $\mathbf{n}_{13}$ gives a 0 for this value, $p(\mathbf{n}_{13})(b)$ remains 0. Now, call *submit*($\mathbf{n}_3, \mathbf{n}_{13}, \{(b)\}, k, k'$) where $\mathbf{n}_3 = \mathit{parent}(\mathbf{n}_{13})$. The call *update*($\mathbf{n}_3, \mathbf{n}_{13}, \{b\}$) gives that $p(\mathbf{n}_3)(b)$ is unchanged (since $p(\mathbf{n}_{13})(b)$ was unchanged). Now, not all possibly affected children of $\mathbf{n}_3$ are done since the leaf $\mathbf{n}_{21}$ representing the formula $x \notin T$ has not yet been propagated. By following a similar reasoning for the nodes $\mathbf{n}_{21}$ and $\mathbf{n}_{22} = \mathit{parent}(\mathbf{n}_{21})$ we will see that the value of $p(\mathbf{n}_{22})(b)$ changes from 1 to 0 (since b is now in T). When this is propagated to $\mathbf{n}_3$ by *submit*($\mathbf{n}_3, \mathbf{n}_{22}, \{(b)\}, k, k'$), the value of $p(\mathbf{n}_3)(b)$ will also change from 1 to 0. A similar reasoning for $\mathit{parent}(\mathbf{n}_3)$, $\mathit{parent}(\mathit{parent}(\mathbf{n}_3))$ and the root node gives the same changes to their penalty mappings consisting of only $() \mapsto 1$. This will lead to an overall penalty decrease of 1 and hence, the penalty of formula (3) with respect to k' is 0, meaning that (3) is satisfied under k'. The values of the changed penalty mappings with respect to k' of $\mathbf{T}$ are shown in parentheses in Figure 2.

6 Neighbourhood Selection

When solving a problem with local search, it is often crucial to restrict the initial configuration and the neighbourhood function used so that not all the constraints need to be stated explicitly. It is sometimes hard by local search alone to satisfy a constraint that can easily be guaranteed by using a restricted initial configuration and neighbourhood function. For example, if a set must have a fixed cardinality, then, by defining an initial configuration that respects this and by using a neighbourhood function that keeps the cardinality constant (for example by swapping values in the set with values in its complement), an explicit cardinality constraint need not be stated. Neighbourhoods are often designed in such an ad-hoc fashion. With the framework of $\exists\text{SOL}^+$, it becomes possible to reason about neighbourhoods and invariants:

Definition 6. *Let formula ϕ model a CSP P, let K be the set of all configurations for P, and let formula ψ be such that $k \models \phi$ implies $k \models \psi$ for all configurations $k \in K$. A neighbourhood function $\mathcal{N} : K \to 2^K$ is* invariant *for ψ if $k \models \psi$ implies $k' \models \psi$ for all $k' \in \mathcal{N}(k)$.*

Intuitively, the formula ψ is implied by ϕ and all possible moves take a configuration satisfying ψ to another configuration satisfying ψ. The challenge then is to find a suitable neighbourhood function for a formula ϕ.

Sometimes (as we will see in Section 7), given formulas ϕ and ψ satisfying Definition 6, it is possible to find a formula δ such that ϕ is logically equivalent to $\delta \wedge \psi$. If the formula δ is smaller than ϕ, then the speed of the local search algorithm can be greatly increased since the incremental penalty maintenance is faster on smaller penalty trees.

7 Application: A Financial Portfolio Problem

After formulating a financial portfolio optimisation problem, we show how to exactly solve real-life instances thereof in our local search framework. This is impossible with the best-known complete search algorithm and competitive with a fast approximation method based on complete search.

7.1 Formulation

The synthetic-CDO-Squared portfolio optimisation problem in financial mathematics has practical applications in the credit derivatives market [7]. Abstracting the finance away and assuming (not unrealistically) interchangeability of all the involved credits, it can be formulated as follows.[1] Let $V = \{1, \ldots, v\}$ and let $B = \{1, \ldots, b\}$ be a set of credits. An *optimal portfolio* is a set of v subsets $B_i \subseteq B$, called *baskets*, each of size r (with $0 \leq r \leq b$), such that the maximum intersection size of any two distinct baskets is minimised.

[1] We use the notation of the related balanced incomplete block design problem.

	c	r	e	d	i	t	s	
basket 1	1	1	1	0	0	0	0	0
basket 2	1	1	0	1	0	0	0	0
basket 3	1	1	0	0	1	0	0	0
basket 4	1	1	0	0	0	1	0	0
basket 5	0	0	1	1	1	0	0	0
basket 6	0	0	1	1	0	1	0	0
basket 7	0	0	1	1	0	0	1	0
basket 8	0	0	0	0	1	1	0	1
basket 9	0	0	0	0	1	0	1	1
basket 10	0	0	0	0	0	1	1	1

Table 1. An optimal solution to $\langle 10, 8, 3, \lambda \rangle$, with $\lambda = 2$.

There is a universe of about $250 \leq b \leq 500$ credits. A typical portfolio contains about $4 \leq v \leq 25$ baskets, each of size $r \approx 100$. Such real-life instances of the portfolio *optimisation* problem are hard, so we transform it into a CSP by also providing a targeted value, denoted λ (with $\lambda < r$), for the maximum of the pairwise basket intersection sizes in a portfolio. Hence the following formulation of the problem:

$$\forall\ i \in V : |B_i| = r \tag{4}$$

$$\forall\ i_1 \neq i_2 \in V : |B_{i_1} \cap B_{i_2}| \leq \lambda \tag{5}$$

We parameterise the portfolio CSP by a 4-tuple $\langle v, b, r, \lambda \rangle$ of independent parameters. The following formula gives an optimal lower bound on λ [13]:[2]

$$\lambda \geq \frac{\lceil \frac{rv}{b} \rceil^2 (rv \bmod b) + \lfloor \frac{rv}{b} \rfloor^2 (b - rv \bmod b) - rv}{v(v-1)} \tag{6}$$

7.2 Using Complete Search

One way of modelling a portfolio is in terms of its *incidence matrix*, which is a $v \times b$ matrix, such that the entry at the intersection of row i and column j is 1 if $j \in B_i$ and 0 otherwise. The constraints (4) and (5) are then modelled by requiring, respectively, that there are exactly r ones (that is a sum of r) for each row and a scalar product of at most λ for any pair of distinct rows. An optimal solution, under this model, to $\langle 10, 8, 3, \lambda \rangle$ is given in Table 1, with $\lambda = 2$.

The baskets are indistinguishable, and, as stated above, we assume that all the credits are indistinguishable. Hence any two rows or columns of the incidence matrix can be freely permuted. Breaking all the resulting $v! \cdot b!$ symmetries can in theory be performed, for instance by $v! \cdot b! - 1$ (anti-)lexicographical ordering constraints [4]. In practice, strictly anti-lexicographically ordering the rows (since baskets cannot be repeated in portfolios) as well as anti-lexicographically

[2] It often improves the bound reported in [7] and negatively settles the open question therein whether the $\langle 10, 350, 100, 21 \rangle$ portfolio exists or not.

ordering the columns (since credits can appear in the same baskets) works quite fine for values of b up to about 36, due to the constraint (5), especially when labelling in a row-wise fashion and trying the value 1 before the value 0. However, this is one order of magnitude below the typical value for b in a portfolio. In [7], we presented an approximate and often extremely fast method of solving real-life instances of this problem by complete search, even for values of λ quite close, if not identical, to the lower bound in (6). It is based on embedding (multiple copies of) independent sub-instances into the original instance. Their determination is itself a CSP, based on (6).

7.3 Using Local Search

It is easy to model the portfolio problem in $\exists\text{SOL}^+$ using additional set variables. The problem can be modelled by the following formula:

$$\exists B_1, \ldots, \exists B_v \exists_{i<j} I_{(i,j)} \ \phi_1 \wedge \phi_2 \wedge \phi_3 \tag{7}$$

where $\exists_{i<j} I_{(i,j)}$ is a shorthand for the sequence of quantifications $\exists I_{(1,2)}, \ldots, I_{(i,j)}, \ldots$ for all $i < j$.[3] The formula $\phi_1 = |B_1| = r \wedge \cdots \wedge |B_v| = r$ states that each set B_i is of size r. Using similar conventions, the formula $\phi_2 = \forall i < j \ \forall x (x \in I_{(i,j)} \leftrightarrow (x \in B_i \wedge x \in B_j))$ states that each set $I_{(i,j)}$ is the intersection of B_i and B_j. Finally, the formula $\phi_3 = \forall i < j |I_{(i,j)}| \leq \lambda$ states that the intersection size of any B_i and B_j should be less than or equal to λ.

The local search algorithm can be made more efficient by using the ideas in Section 6. First, we define a neighbourhood function that is invariant for the formula ϕ_1. Assuming that the initial configuration for (7) respects ϕ_1, the neighbourhood function that swaps any value in any B_i to any value in its complement is invariant for ϕ_1. We denote this neighbourhood function by *exchange*. We may even extend *exchange* such that it is invariant also for ϕ_2. In order to do this, we assume that the initial configuration for (7) respects $\phi_1 \wedge \phi_2$. Now, we extend *exchange* in the following way. Given a configuration k and a configuration k' in $exchange(k)$ where B_i is the only variable affected by the change of k to k', the variables $I_{(i,j)}$ such that there exists a subformula $x \in I_{(i,j)} \leftrightarrow (x \in B_i \wedge x \in B_j)$ or $x \in I_{(j,i)} \leftrightarrow (x \in B_j \wedge x \in B_i)$ are all updated (by adding or removing a value to $I_{(i,j)}$) so that those formulas still hold.

We use a similar algorithm to the one in [2] for solving the portfolio problem with local search, i.e., a Tabu-search algorithm with a restarting criterion if no overall improvement was reported after a certain number of iterations.

7.4 Results

The experiments were run on an Intel 2.4 GHz Linux machine with 512 MB memory. The local search framework was implemented in OCaml and the complete search algorithm was coded in SICStus Prolog.

[3] This shorthand is a purely conservative extension of $\exists\text{SOL}^+$ and does not increase the expressiveness.

The local search algorithm performs well on this problem. For example, the easy instance $\langle 10, 35, 11, 3\rangle$ is solved in 0.2 seconds, the slightly harder instance $\langle 10, 70, 22, 6\rangle$ in 0.6 seconds, and the real-life instance $\langle 15, 350, 100, 24\rangle$ in 133.9 seconds. Bear in mind that these results were achieved (by our current prototype implementation) under the assumption that no built-in constraints existed, and thus that the incremental penalty maintenance algorithms were automatically generated as described in this paper.

For comparison, the complete search approach without embeddings needs 0.6 seconds for finding a first solution of $\langle 10, 35, 11, 3\rangle$, 929.8 seconds for $\langle 10, 70, 22, 6\rangle$, and does not terminate within several hours of CPU time for $\langle 15, 350, 100, 24\rangle$.

Using the extended implementation [13] of the embedding method of [7] for the real-life instance $\langle 15, 350, 100, 24\rangle$, two embeddings were constructed but both timed out after 100 seconds. Hence, local search approaches can outperform even this approximation method.

8 Conclusion

Summary. In the context of local search, we have introduced a scheme that, from a high-level problem model in existential second-order logic with counting ($\exists$SOL$^+$), automatically synthesises incremental penalty calculation algorithms. This bears significant benefits when ad hoc constraints are necessary for a particular problem, as no adaptation by the user of the modelling part of the local search system is then required. The performance of the scheme has been demonstrated by solving real-life instances of a financial portfolio design problem that seem unsolvable in reasonable time by complete search.

Related Work. The usage of existential second-order logic ($\exists$SOL) as a modelling language has also been advocated in [9]. The motivation there was rather that any automated reasoning about constraint models must *necessarily* first be studied on this simple core language before moving on to extensions thereof. Modern, declarative constraint modelling languages, such as NP-SPEC [3], OPL [14], and ESRA [6], are extensions of $\exists$SOL. In contrast, our motivation for $\exists$SOL is that it is a *sufficient* language for our purpose, especially if extended (only) with counting.

The adaptation of the traditional combinators of constraint programming for local search was pioneered in [15]. The combinators there include logical connectives (such as $\wedge$ and $\vee$), cardinality operators (such as *exactly* and *atmost*), reification, and expressions over variables. We extend these ideas here to the logical quantifiers ($\forall$ and $\exists$). This is not just a matter of simply generalising the arities and penalty calculations of the $\wedge$ and $\vee$ connectives, respectively, but made necessary by our handling of set variables over which one would like to iterate, unlike the scalar variables of [11, 15].

Future Work. We have made several simplifying assumptions in order to restrict this paper to its fundamental ideas. For instance, the handling of both scalar variables and set variables requires special care in the calculation of penalties, and has been left as future work. Also, many more shorthand notations than

the ones used in this paper could be added for the user's convenience, such as quantification bounded over a set rather than the entire universe. Furthermore, it would be useful if appropriate neighbourhood functions that are invariant for some of the constraints could automatically be generated from an $\exists\text{SOL}^+$ model.
Conclusion. Our first computational results are encouraging and warrant further research into the automatic synthesis of local search algorithms.
Acknowledgements. This research was partially funded by Project C/1.246/HQ/JC/04 of EuroControl. We thank Olof Sivertsson for his contributions to the experiments on the financial portfolio problem, as well as the referees for their useful comments.

References

1. E. Aarts and J. K. Lenstra, editors. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997.
2. M. Ågren, P. Flener, and J. Pearson. Set variables and local search. In *Proc. of CP-AI-OR'05*, volume 3524 of *LNCS*, pages 19–33. Springer-Verlag, 2005.
3. M. Cadoli, L. Palopoli, A. Schaerf, and D. Vasile. NPSPEC: An executable specification language for solving all problems in NP. In *Proc. of PADL'99*, volume 1551 of *LNCS*, pages 16–30. Springer-Verlag, 1999.
4. J. M. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. of KR'96*, pages 148–159. Morgan Kaufmann, 1996.
5. R. Fagin. *Contributions to the Model Theory of Finite Structures*. PhD thesis, UC Berkeley, California, USA, 1973.
6. P. Flener, J. Pearson, and M. Ågren. Introducing ESRA, a relational language for modelling combinatorial problems. In *LOPSTR'03: Revised Selected Papers*, volume 3018 of *LNCS*, pages 214–232. Springer-Verlag, 2004.
7. P. Flener, J. Pearson, and L. G. Reyna. Financial portfolio optimisation. In *Proc. of CP'04*, volume 3258 of *LNCS*, pages 227–241. Springer-Verlag, 2004.
8. N. Immerman. *Descriptive Complexity*. Springer-Verlag, 1998.
9. T. Mancini. *Declarative constraint modelling and specification-level reasoning*. PhD thesis, Università degli Studi di Roma "La Sapienza", Italy, 2004.
10. L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. In *Proc. of CP'97*, volume 1330 of *LNCS*, pages 237–251. Springer-Verlag, 1997.
11. L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. *ACM SIGPLAN Notices*, 37(11):101–110, 2002. Proc. of OOPSLA'02.
12. A. Nareyek. Using global constraints for local search. In *Constraint Programming and Large Scale Discrete Optimization*, volume 57 of *DIMACS: Series in Discrete Mathematics and Theoretical Computer Science*, pages 9–28. American Mathematical Society, 2001.
13. O. Sivertsson. Construction of synthetic CDO Squared. Master's thesis, Computing Science, Department of Information Technology, Uppsala University, Sweden, 2005.
14. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.
15. P. Van Hentenryck, L. Michel, and L. Liu. Constraint-based combinators for local search. In *Proc. of CP'04*, volume 3258 of *LNCS*, pages 47–61. Springer-Verlag, 2004.
16. J. P. Walser. *Integer Optimization by Local Search: A Domain-Independent Approach*, volume 1637 of *LNCS*. Springer-Verlag, 1999.

Recent licentiate theses from the Department of Information Technology

2003-011 Tobias Amnell: *Code Synthesis for Timed Automata*

2003-012 Olivier Amoignon: *Adjoint-Based Aerodynamic Shape Optimization*

2003-013 Stina Nylander: *The Ubiquitous Interactor - Mobile Services with Multiple User Interfaces*

2003-014 Kajsa Ljungberg: *Numerical Methods for Mapping of Multiple QTL*

2003-015 Erik Berg: *Methods for Run Time Analysis of Data Locality*

2004-001 Niclas Sandgren: *Parametric Methods for Frequency-Selective MR Spectroscopy*

2004-002 Markus Nordén: *Parallel PDE Solvers on cc-NUMA Systems*

2004-003 Yngve Selén: *Model Selection*

2004-004 Mohammed El Shobaki: *On-Chip Monitoring for Non-Intrusive Hardware/Software Observability*

2004-005 Henrik Löf: *Parallelizing the Method of Conjugate Gradients for Shared Memory Architectures*

2004-006 Stefan Johansson: *High Order Difference Approximations for the Linearized Euler Equations*

2005-001 Jesper Wilhelmsson: *Efficient Memory Management for Message-Passing Concurrency — part I: Single-threaded execution*

2005-002 Håkan Zeffer: *Hardware-Software Tradeoffs in Shared-Memory Implementations*

2005-003 Magnus Ågren: *High-Level Modelling and Local Search*

UPPSALA
UNIVERSITET

Department of Information Technology, Uppsala University, Sweden