Constraints over structured domains

Carmen Gervet

Chapter 17 Constraints over Structured Domains

Carmen Gervet

The computer will be the most marvellous
of all tools as soon as program writing and
debugging will be no longer necessary
—Jean-Louis Laurière (1976)

A wide range of combinatorial search problems find a natural formulation in the language of sets, multisets, strings, functions, graphs or other structured objects. Bin-packing, set partitioning, set covering, combinatorial design problems, circuits and mapping problems are some of them. They are NP-complete problems originating from areas as diverse as combinatorial mathematics, operations research or artificial intelligence. These problems deal essentially with the search for discrete structured objects. While a high-level modeling approach seems more natural, many solutions have exploited the effectiveness of finite domains or mixed integer programming solvers. In this chapter we present higher level modeling facilities utilizing constraints over structured domains.

What is a structured object? Let us consider the example of a bin-packing problem. The main constrained objects are the different bins, each describing a collection of unordered distinct elements, subject to disjointness constraints among them, weight constraints reflecting on each bin capacity and possible cardinality restrictions on the number of items allowed in each bin. Informally, such objects are structured in the sense that they involve more than one element *in a specific setting*.

When Fikes introduced the notion of finite domain in 1970 [31], the idea was to approximate the range of an unknown integer (an integer variable) and to prune inconsistent values from such a domain that cannot belong to any solution. Already in the description of the language REF-ARF, Fikes proposed directions for future work such as: "*considering the addition to the program of capabilities for handling unordered sets*". Mid-eighties the seminal work of Van Hentenryck et al. integrated consistency techniques over finite integer domains into logic programming [90], and gave birth to the first finite domain constraint logic programming language CHIP (Constraint Handling In Prolog) [23], leading to a new

generation of academic and industrial constraint programming systems. The successes of CHIP and its peers also raised the questions of the languages limitations. While Finite Domain (FD) solvers grew in efficiency, it remained that models lacked generic and natural formulations when representing structured objects, making the programming effort more cumbersome and sometimes ad-hoc.

As an example, let us consider the structured object, "set", constrained to be subset of a known base set. A finite domain approach would consider two possible representations:

- a list of FD variables taking their value from a finite set of integers that represents the base set. This approach requires the removal of order and multiplicity among the elements of the list, which is achieved by adding ordering constraints. For example the list $[X_1, X_2, X_3] :: [0..5], X_1 < X_2 < X_3$ represents a set of 3 elements subset of the set $\{0, 1, 2, 3, 4, 5\}$. If the size of the set is unknown some dummy FD variables are also necessary. Clearly this does not make easy the modeling of additional set constraints such as intersection, or union.

- a list of 0-1 FD variables. This second formulation is equivalent to the semantics of a finite set subset of a known set. It uses 0-1 variables, and originates from 0-1 Integer Programming (ILP). Basically, this approach exploits the one-to-one correspondence that exists between a subset s of a known set S and a Boolean algebra. The correspondence is defined by the characteristic function:

  $$f : y_i \rightarrow \{0, 1\} \;\; f(y_i) = 1 \text{ iff } i \in s$$

  In other words, a 0-1 variable is associated with each element in the base set S and takes the value 1 if and only if the element belongs to the unknown set s. Set constraints are then simply represented within the Boolean algebra with arithmetic operators. The main drawback of this representation is that it looses the semantics and structure of the problem addressed. Operationally it can benefit from global reasoning from mathematical programming, but in a constraint programming environment lacks conciseness and does not best exploit the problem structure. We give further on some comparisons between 0-1 and set models.

Approach. This chapter is not intended to give a complete coverage of all results available in softwares and systems that embed constraints over structured domains. Instead we try to cover a number of significant research topics in more detail. This should give a context and picture for the research and its methodology, provide the most important references, and enable the reader to study research papers on the topic.

17.1 History and Applications

Before the research field of "constraint programming" even existed, the seminal work of Laurière in 1976 proposes constrained structured objects in the design and development of `ALICE` [61]. Laurière's idea was to combine generality and efficiency in addressing combinatorial problems. He defines an input language, purely descriptive, with high level objects such as functions and relations between two known sets, constrained by some properties, such as injection and bijection. `ALICE` was a pioneer in the use of structured objects to model combinatorial problems.

In a parallel line of research, most proposals to extend constraint reasoning over new structured domains came as extensions of the logic programming system, Prolog. Logic programming is a powerful programming framework which enables the user to state non-deterministic programs in relational form [56, 20]. The extension to Constraint Logic Programming (CLP) combines the positive features of logic programming with constraint solving techniques, where the concept of constraint solving replaces the unification procedure in logic programming and provides, among others, a uniform framework for handling new structured domains. Previous chapters have presented in depth the state of the art in the precursors constraint domains (rationals, Boolean algebra, finite integer domains, real intervals). This chapter is interested in discrete and structured domains such as strings, finite sets in different forms, relations, maps and graphs.

In 1989 Walinsky presented CLP(Σ^*), an instance of the CLP scheme over the computation domain of strings, represented as regular sets [92]. The practical motivation was to incorporate strings into logic programming to strengthen the standard string-handling features (e.g. concat, substring). It constitutes the first attempt to compute regular sets by means of constraints like the membership relation. For example `A in (X.''ab''.Y)` states that any string assigned to variable `A` must contain the substring `ab`. This approach was further developed by Golden and Pang in 2003 [42] even though they did not seem aware of Walinsky's work. Their main contribution is to use finite automata to represent regular sets. Both approaches consider possibly infinite sets of strings. More recently Pesant proposed a global constraint on a fixed length sequence of finite domain variables with application to rostering and car sequencing problems [74].

The most widely studied structured domain is most definitely that of sets. The motivations to embed sets in constraint based languages are quite diverse and address different issues ranging from program analysis, software prototyping and specification, set theory axiomatization and combinatorial problem solving. The terminology of "set constraints" is worth a few words even though it does not relate directly to constraint satisfaction problems. Heintze and Jaffar [46] coined the term of set constraints in 1990 to handle a class of sets of trees (possibly infinite) and to deal with relations of the form $s_1 \subseteq s_2$ where s_1 and s_2 denote specific set expressions, possibly recursive, defined over trees. This line of research applies to program analysis systems ([6, 2] among others) which was pioneered by John Reynolds in 1969 [81]. Besides the terminology of set constraints, these systems do not relate to constraint programming over a specific computation domain, as they do not interpret set operations but rather show the expressiveness of "set constraints" for the analysis of programs developed in logic or functional programming. For further information, please refer to survey articles such as [72].

Ironically it was about the same time that the notion of *finite sets* was embedded as a high level programming abstraction in logic-based and then constraint (logic) based languages, in quite a different setting. We refer to the term set to denote a finite set. A set is basically a collection of distinct elements commonly described by $\{x_1, ..., x_n\}$. The use of a logic-based language as the underlying framework came from proposals in database query languages where the aim was to strengthen typical existing set facilities of languages like Prolog (e.g. `setof, bagof`) to handle sets of terms and complex data structures. In this line of work sets have been embedded in [10, 57, 87, 25]. All these languages converge on one aspect: representing a set variable by a set constructor so as to nest objects in a natural manner. This constructor is specified either by an extensional representation $\{x_1, ..., x_n\}$ ([10, 57]) or by an iterative one $\{x\} \cup E$ where E can be unified with a set of

terms containing possibly set variables (concept of sets of finite depth, or hereditarily finite sets in $\{\log\}$ [25], CLPS [63], and [89]).

Even though these languages use constraints to reason upon sets, they do face the NP-completeness of the equality relation over constructed sets (as a particular case of Associative, Commutative and Idempotent (ACI) relation [65]). The main reason is the absence of a unique most general unifier when unifying constructed sets. For example, the equality $\{X, Y\} = \{3, 4\}$ derives two solution sets: $\{X = 3, Y = 4\}$ and $\{X = 4, Y = 3\}$ neither of which is more general than the other. This means that the satisfaction of the ACI axioms, introduces nondeterminism in the unification procedure by deriving disjunctions of a finite number of equalities.

While such approaches did not offer a practical solution to set unification they were not essentially motivated by effective solving of combinatorial problems. In 1992, a new class of finite set constraint solvers was designed to expand the modeling facilities of finite domain solvers when tackling set-based combinatorial search problems (e.g. bin packing, set partitioning, combinatorial designs or more recently network design). The idea was developed independently by Puget [75, 76], and Gervet [37, 38]. The objective was to propose a high-level modeling language which enabled us to model a set-based combinatorial problem as a set domain CSP – where set variables range over set intervals – and which tackled set constraints by using consistency techniques. A set domain is a collection of known sets of arbitrary elements like $\{\{1, 2\}, \{1, 4\}, \{1, 5\}\}$. It is specified by a set interval, $[\{1\}, \{1, 2, 4, 5\}]$, where the lower bound contains the *definite* elements of the set, and the upper bound extends it with possible elements. Gervet formalized the concepts and ideas when presenting the `Conjunto` language in [39, 40]. Though implementation details vary, at their core the set constraint solvers of `solver` [77], ECL^{i}PS^e[86], `MOZART-OZ` [67, 69], `FACILE` [7], `B-Prolog` [94], `CHOCO` [58], all have the subset bounds as domain representation. The availability of all these solvers both in academia and industry, has enabled the design of new models and solutions to problems from combinatorial mathematics [8], VLSI circuit verification and warehouse location [3], as well as network design problems (e.g. weight setting [29], SONET [88, 85]).

However, it has also raised the question of the limitations of the core `Conjunto`-like set interval solver, leading to further research in this area. Research advances in finite set solvers include: i) the extension of the core subset bound solver with new inferences relative to the set cardinality constraint (mainly described in `Cardinal` [3] and `OZ` [68]), ii) the development of global set constraint propagators, iii) the search for more expressive set domain representations.

Regarding global set constraints propagators, Sadler and Gervet investigated the case of n-ary constraints on fixed cardinality sets such as `atmost1, distinct`, stating respectively that n sets of known cardinality should intersect pairwise in atmost one element, or not be equal [82]. This first attempt was followed by challenging results both theoretically and algorithmically. Walsh in [93] addresses the question of whether such global constraints could infer anything more than their decompositions and with Bessière et al. started a systematic investigation of determining the tractability of a range of global set constraints [14]. New global propagators were presented for the `disjoint` and `partition` constraints for sets of known cardinality, independently by Sadler and Gervet in [83] and Bessière et al. in [13]. Such constraints have been present in Ilog `SOLVER` with similar algorithms [50].

Regarding the effectiveness of finite set intervals, Lagoon and Stuckey propose in [60]

a radically different approach to the standard subset domain bounds. They show that Reduced Ordered Binary Decision Diagrams (ROBDDs) can be used to represent full domains efficiently. The same year, the set interval representation was also reconsidered by Sadler and Gervet in [84] in order to make better use of the cardinality information and break set symmetries in problems such as combinatorial designs [85]. They define a hybrid set domain whereby the conventional subset domain is enriched with a lexicographic domain that shows to better exploit the cardinality information and symmetry breaking constraints. As this chapter was compiled, Gervet and Van Hentenryck proposed a length-lex representation of set domains that encodes directly cardinality and lexicographic information, and shows promise in reaching powerful and cost effective pruning [41].

Other structured objects have been considered to expand the modeling facilities of finite domain constraints. Multisets (sets where an element may occur more than once), commonly referred to as *bags*, have been embedded in few constraint languages and seem an adequate choice of model for template design problems [54]. Existing approaches to multiset reasoning make use of constructors or domains. For example `CLPS` uses multiset constructors while `SOLVER` uses multiset domains. In [93], Walsh formalizes the idea of multiset domains and discusses the expressiveness of different domain representations. Quimper and Walsh also recently proposed in [78] to use efficient enumeration procedures (see Knuth [55]) to extend the use of some global constraint on large domains over sets, but also tuples represented as lists of integer variables.

Finally, higher level structured domains have recently been re-discovered (graph and map variables) or proposed (ontologies, lattices). The proposals follow two main trends: i) high level constructors that are part of a specification or modelling language compiled into an executable code such as the works of Flener et al. [32] leading to the modeling language `ASRA` [32], and the PhD thesis of Hnich in $\mathcal{L}$ [48], ii) high level computation domains to reason with and about relations and graphs as in `Conjunto`[39], `CP(Graph)` [24], and `CP(Graph + Map)`[22], and order-sorted domains introduced by Caseau and Puget [17], as well as ontology domains introduced by Laburthe [59]. Fernández and Hill generalized all interval reasoning approaches over structured domains that are lattices into a single framework, deriving the $clp(\mathcal{L})$ language [30].

17.2 Constraints over Regular and Constructed Sets

Most of the recent proposals (late eighties) to embed strings or constructed sets as a high level programming abstraction aim at extending a logic-based language and thus assume such a language as the underlying framework. In this section we review the major approaches which embed strings and constructed sets in constraint programming.

17.2.1 Regular Sets

CLP(Σ^*). This language represents an instance of the CLP scheme over the computation domain of regular sets[92]. A regular set is a finite set composed of strings which are generated from a finite alphabet Σ. CLP(Σ^*) has been designed and implemented to provide a logic-based formalism for incorporating strings into logic programming in a more expressive manner than the standard string-handling features (eg. `concat, substring`). A CLP(Σ^*) program is a Prolog program enriched with regular set terms and built-in constraints.

Operations on regular sets comprise concatenation $R_1.R_2$, disjunction or union $R_1 + R_2$ (i.e., $R_1 \cup R_2$) and the closure operator R_1^* which describes the least set R' such that $R' = \epsilon + (R', R_1)$. These operations allow us to build any regular expression when combined with the identity elements under concatenation (1) and union ($\emptyset$). This language provides an atomic constraint over set expressions which is the membership constraint of the form x `in` e where x is either a variable or a string and e is a regular expression. For example A `in` $(X.''ab''.Y)$ states that any string assigned to variable A must contain the substring ab.

The satisfiability of membership constraints over regular sets clearly poses the problem of termination. In the above example, if Y is a free variable there is an infinite number of instances for A. The solver guarantees termination by: (i) applying a scheduling strategy which selects the constraints capable of generating a finite number of instances, (ii) applying a satisfiability procedure based on deduction rules which check and transform the selected atomic constraints. The non selected ones are simply floundered.

The selected constraints x `in` e are such that either e is a string or e is a variable and x a string. The conditional deduction rules over each of these constraints infer a new constraint or a simplified one if a given condition is satisfied. Each condition represents a possible form of selected set constraints.

As an example, the following rules describe the derivation of concatenated expressions under idempotent substitutions:

$$\frac{\left(\begin{array}{c} w = w_1.w_2 \\ \sigma_1 \vdash\ ''w_1'' \texttt{ in } e_1 \\ \sigma_2 \vdash\ ''w_2'' \texttt{ in } e_2 \end{array}\right)}{\sigma_1 \cup \sigma_2 \vdash\ ''w'' \texttt{ in } e_1.e_2} \quad \text{and} \quad \frac{\left(\begin{array}{c} \sigma_1 \vdash X_1 \texttt{ in } e_1 \\ \sigma_2 \vdash X_2 \texttt{ in } e_2 \end{array}\right)}{[X = (X_1\sigma_1).(X_2\sigma_2)] \vdash X \texttt{ in } e_1.e_2}$$

The σ_i are idempotent substitutions, which means that given two substitutions σ_1 and σ_2, $\sigma_1 \cup \sigma_2$ produces the most general idempotent substitution if one exists that is more specific than the two previous ones.

Soundness and completeness of the deduction rules are guaranteed only if there are no variables within the scope of any closure expression e^* in addition to the criteria of constraint selection. This approach constitutes a first attempt to compute regular sets by means of constraints like the membership relation. The complexity of the satisfiability procedure is not given, but infinite computations are avoided thanks to the use of floundering.

Regular sets and finite automata. The key challenges when reasoning about string constraints effectively are 1) to represent infinite string sets without actually requiring infinite space, and 2) to enforce constraints over infinite string sets without exhaustively listing the consistent values [42]. To do so one would use regular languages, i.e. sets of strings accepted by regular expressions or finite automata, which are widely used for instance in string matching or lexical analysis.

Constraints over the string variables extend the ones presented in CLP(Σ^*) with constraints on the length of a string `length`. Two different representations of regular languages are used: regular expressions and finite automata (FAs) [49]. Regular expressions that represent a regular language over an alphabet Σ, are used as input and are converted to FAs, which are used computationally. This system has been used within a constraint

based planner for NASA. The solver performs set operations on Finite Automata to prune the string domains and reach a consistent state. All of the set operations and string constraints are either linear or quadratic in the size of the FAs representing the string domain. However, the FA can grow exponentially with the number of operations, i.e. the number of constraints that contain the variable whose domain is represented by the FA. Ultimately how the FA grows will depend on the nature of the problem at hand.

Such languages allow variables to range over an infinite set of strings. This is suitable for their motivational problems but is not a requirement in all application domains involving strings.

The use of membership constraints for sequences of finite domain variables also exists in the constraint programming literature to address in particular combinatorial search problems such as rostering and car sequencing. The objective is usually to identify or enforce patterns of values, specified over finite domain variables. The approaches are commonly embedded as global constraints with associated propagator. We refer the reader to the `sequence` constraint (constrains the number of times a certain pattern of length l appears in a sequence of variables) introduced in [80], solver's `IlcTableConstraint` [50] (takes a sequence of n finite-domain variables and a set of n-tuples representing the valid assignments of values to these variables), or the more recent `regular(x,M)` constraint [74]. This constraint is a regular language membership constraint that constrains "any sequence of values taken by the finite domain variables of x to belong to the regular language recognized by M". It reasons upon strings of the regular language that have a given length n which is powerful enough for its purpose.

The embedding and use of regular sets in constraint (logic) programming has a clear diversity from enhancing the string manipulation of Prolog to enforcing patterns of values in combinatorial search problems.

17.2.2 Constraints over Constructed Sets

The first steps towards embedding sets in constraint programming first assumed a logic-based language as the underlying framework. This follows from the declarative nature of logic programming, which well combines with set constructs, and its nondeterminism which is suited to stating set-based programs. The presented languages are the main ones relating to constraint reasoning. More literature exists relating solely to logic programming.

{log} and CLP($\mathcal{SET}$). {log} [25, 26, 27] has been designed and implemented mainly for theorem proving. Consequently, it embeds an axiomatized set theory whose properties guarantee soundness and completeness of the language.

Set terms are constructed using the interpreted functors `with` and {}, e.g. $\emptyset$ `with` x `with` ($\emptyset$ `with` y `with` z) = {{z,y},x}. The language includes a limited collection of predicates ($\in, =, \neq, \notin$) as set constraints. The axiomatized set theory consists of a set of axioms which describe the behaviour of the constructor *w*ith. For example the *extensionality axiom* shows how to decide if two sets can be considered equal:

$$v \texttt{ with } x = w \texttt{ with } y \rightarrow \\ (x = y \wedge v = w) \vee (x = y \wedge v \texttt{ with } x = w) \vee \\ (x = y \wedge v = w \texttt{ with } y) \vee \exists z\, (v = z \texttt{ with } y \wedge w = z \texttt{ with } x)$$

Using the axioms, a set of properties are derived describing the permutativity (right associativity) and absorption of the `with` constructor. For example, the permutativity property is depicted by:

$(x \texttt{ with } y) \texttt{ with } z = (x \texttt{ with } z) \texttt{ with } y$ (permutativity)

The complete solver consists of a constraint simplification algorithm defined by a set of derivation rules with respect to each primitive constraint. A derivation rule for the equality constraint is, for example:

$h \texttt{ with } \{t_n, ..., t_0\} = k \texttt{ with } \{s_m, ..., s_o\}$

If h and k are not the same variables then select non-deterministically one action among a set of possible substitutions (minimal set of unifiers). The nondeterministic satisfaction procedure of constructed sets reduces a given constraint to a collection of constraints in a suitable form by introducing choice points in the constraint graph itself. This leads to a hidden exponential growth in the search tree. In this approach, completeness of the solver is required if one aims at performing theorem proving. Thus, there is no possible compromise here between completeness and efficiency. The soundness and completeness of its solver allow us to use it for theorem proving and problem specification.

$\{\log\}$ has been revisited from a LP to a CLP framework in order to provide a uniform framework for the handling of set constraints ($\in, =, \neq, \notin$). The CLP counterpart called CLP($\mathcal{SET}$) is described in [28]. The design and implementation of $\{\log\}$ and subsequently CLP($\mathcal{SET}$) have settled the theoretical foundations for embedding constructed sets of the form $\{x\} \cup S$ into (constraint) logic programming.

CLPS. The CLPS language (Constraint Logic Programming with Sets) was designed for prototyping combinatorial search problem dealing with sets, multisets, or sequences. It is based on a three sorted logic, the three sorts being: sets, multisets and sequences of finite depth (eg. $s = \{\{\{e, a\}\}, c\}$ is a set of depth three) [63]. The concept of depth is equivalent for each sort.

In CLPS, set expressions are built from the usual set operator symbols ($\cup, \cap, \setminus, \#$). Set variables are constructed either iteratively by means of the set constructor $\{x\} \cup s$ or by extension by grouping elements within braces (eg. $\{x_1, ..., x_n\}$). The language also embeds finite integer domains and allows set elements to range over a finite domain. Sequences and multisets are built using, respectively, the constructors $sq\{...\}$ and $m\{...\}$. Basic constraints are relations from $\{\in, =, \notin, \neq, \subseteq\}$ interpreted in the usual mathematical way together with a depth (::) and a type checking operator.

The satisfiability problem for sets, sequences and multisets is $\mathcal{NP}$-complete [65]. To cope with this, CLPS provides several methods whose use depends on the characteristics of the CLPS program at hand. The solver makes use of various techniques comprising: (i) a set of semantical-consistency rules, (ii) an arc-consistency algorithm of type AC-3 [66] combined with a local search procedure (forward checking) and (iii) a transformation procedure which transforms the set constraint system into an equivalent mathematical model based on integer linear programming [47]. The rules in (i) check the consistency of each set constraint with respect to homogeneity of types, depth and cardinality. For example the system

$\{x\} = \{y, z\}$ is semantically-consistent if $y = z$

A semantically-consistent system of set constraints is then solved in two stages. The solver first divides the system in two independent subsets: 1) the first one, SC_{fd}, contains set constraints whose constrained sets are sets of integer domain variables, 2) the other one, written SC_v contains sets and set constraints where set elements are free variables or known values. The solver applies (ii) and (iii) respectively to check satisfiability over SC_{fd} and SC_v.

An interesting component is the resolution of SC_v using (iii). A system SC_v is satisfiable if its equivalent integer linear programming form is satisfiable [47]. To check satisfiability, the system provides a correct and complete procedure which transforms the set constraint system into an equivalent mathematical model based on integer linear programming. This procedure consists in flattening each set constraint and reducing the system of flattened formulas to an equivalent system of linear equations and disequations over finite domain variables. The derived system is then solved using consistency techniques. The flattening algorithm works by adding additional variables to reach forms from ($x = y,\ x \in y,\ x = \{x_1, ..., x_n\}, x = y \cup z,\ x = y \cap z,\ x = y \setminus z$, etc.). The reduction to linear form is performed by associating to each set variable x_i a new variable C_{xi} which represents its cardinality and to each pair of variables $(x_i,\ x_j)$ a new binary variable Q_{ij} denoting possible set equality constraints. If there are n constraints the complexity of the reduction procedure is in $\mathcal{O}(n^3)$.

The proposed solving methods are among the most appropriate for handling set constraints over constructed sets. They fit the application domain of the language which aimed initially at combinatorial problem prototyping. Unfortunately the nondeterminism in the unification of set/multisets/strings constructs prevents an efficient pruning of the domains attached to set elements (in case they represent domain variables). The focus is put on the expressive power of the language rather than on the efficient solving.

Since its first release, the CLPS kernel has been extended in many ways. In particular, new solvers on constructed terms for multisets and sequences have been defined based on PQR-trees and proved to be appropriate for modelling and solving scheduling problems with a reasonable efficiency [9]. The application domain of CLPS has since migrated and a new solver called CLPS-B has been designed and implemented to animate and generate test sequences from B and Z formal specifications [15]. The B method, developed by Abrial, forms part of a formal specification model based on first order logic extended to set constructors and relations, (see [1] for a description of the B method).

17.3 Constraints over Finite Set Intervals

As we mentioned earlier on, many combinatorial search problems find a natural formulation in the language of sets. The embedding of finite set intervals in constraint programming languages builds upon the successes of finite domain constraint satisfaction problem (CSP) in order to allow for natural and concise modeling of a set-based combinatorial search problems as set domain CSP – where set variables range over finite set domain – and set constraints are handled using consistency techniques. The motivations differ slightly from the previous languages since the approach compromises expressiveness (sets don't contain variables) with efficiency (trivial deterministic unification of finite sets). We present the main components of the finite set solver, since it is available in most CP lan-

guages and lead to much further research and improvements in recent years. Comprehensive theoretical and practical descriptions can be found in [39, 77, 40].

Notations. Set variables will be represented by the letters x, y, z, s, set constants by the letters a, b, c, d, natural numbers by the letters m, n and integer variables by v, w. All these symbols can be subscripted.

17.3.1 Subset Domain Bounds and Convex Closure Operator

A set domain can be specified in extension as a collection of known sets of arbitrary elements like $\{\{a, b\}, \{c, d\}, \{e\}\}$. However, such domains can be large (e.g., if $s \subseteq \{1, \ldots, 100\}$, its domain contains 2^{100} elements). A common approach to tackling large domains is to approximate the domain reasoning by an interval reasoning as in many FD solvers. This is why the notion of set domain has been approximated by a *set interval specified by its upper and lower bounds*, defined by some appropriate ordering on the domain values. In this case the partial ordering under set inclusion is considered. This enables the use of consistency techniques [66] by reasoning in terms of interval variations, when dealing with a system of set constraints. The set interval $[\{\}, \{a, b, c, d, e\}]$ represents the convex closure of the set domain above.

The core idea is to approximate the domain of a set variable by a closed interval denoted $[glb, lub]$, specified by its unique least upper bound glb, and unique greatest lower bound lub, under set inclusion. Any such interval within a powerset lattice is necessarily convex allowing us to perform correct computations over the set intervals. This approach finds similarities with other interval reasoning approaches like real intervals or Booleans (see [71, 11]).

The glb of the set domain contains the *definite* elements of s and the lub contains in addition *possible* elements of s.

Example 17.1. The constraint $s \in [\{3, 1\}, \{3, 1, 5, 6\}]$ means that the elements $3, 1$ belong to s and that 5 and 6 are possible elements of s.

Regarding set expressions, the domain of a union or intersection of sets is not a set interval because it is not a convex subset of the $\mathcal{P}(\{1, 2, 3, 4, 5, 6\})$, the domain of discourse (e.g. $I = [\{1\}, \{1, 3\}] \cup [\{\}, \{2, 6\}]$, $\{1, 3\}, \{6\} \in I$ but $[\{\}, \{1, 3, 6\}] \not\subseteq I$). It is possible to maintain such disjunctions of domains during the computation, but this leads to a combinatorial explosion. This handling of "holes" can be avoided by considering the convex closure of a set expression domain. To do so one needs a convex closure operation over a subset of a powerset lattice equipped with set inclusion ordering.

Convex closure operation. Let $\mathcal{D}_S$ be the powerset lattice $\langle \mathcal{P}(\mathcal{H}_u), \subseteq \rangle$ with the partial order $\subseteq$ where $\mathcal{P}(s)$ denotes the powerset of s and the universe of discourse $\mathcal{H}_u$ refers to the Herbrand universe. To ensure that any set domain is a set interval, we define a convex closure operation which associates to any $\mathcal{D}_S$ its convex closure as being a set interval.

Definition 17.2. *Given any subset $x = \{a_1, ..., a_n\}$ of $\mathcal{D}_S$ we have:*

$$co\vec{n}v\ (x) = \overline{x} = [\bigcap_{a_i \in x} a_i,\ \bigcup_{a_i \in x} a_i]$$

The convex closure of the set$\{\{3,2\},\{3,4,1\},\{3\}\}$ belonging to $\mathcal{P}(D_S)$ is the set interval $[\{3\},\{1,2,3,4\}]$.

The operations $\bigcap_{a_i \in x} a_i$ and $\bigcup_{a_i \in x} a_i$ derive respectively $glb(x)$and $lub(x)$. The operation $\vec{conv}(x) = \overline{x} = [glb(x), lub(x)]$ satisfies the properties of extension ($x \subseteq \overline{x}$), idempotence ($\overline{x} = \overline{\overline{x}}$), and monotony (if $x \subseteq y$, then $\overline{x} \subseteq \overline{y}$)

The existence of limit elements for any set $\{a, b\}$ belonging to $\mathcal{D}_S$ allows us to define a notion of set domain as a convex subset of $\mathcal{D}_S$, that is a set interval $[a \cap b, a \cup b]$.

Set interval calculus. The powerset algebra $\mathcal{D}_S$ interprets the set function symbols $\cup$, $\cap$, $\setminus$ in their usual set theoretical sense (*i*.e., $\emptyset$ is the empty set, $\setminus$ the set difference, etc.). The interpreted set union and intersection symbols have the usual algebraic properties (commutativity, associativity, idempotence, absorption). By making use of the convex closure operation we ensure that the union and intersection of set intervals yield intervals as well. The resulting set interval calculus is described as follows:

$\overline{[a,b] \cup [c,d]} = [a \cup c, b \cup d]$
$\overline{[a,b] \cap [c,d]} = [a \cap c, b \cap d]$
$\overline{\mathcal{P}(D_s)} = \mathcal{P}(D_s)$ and $\overline{\emptyset} = \emptyset$

With regard to the set difference operation $[a,b] \setminus [c,d]$, its set theoretical definition is $x \setminus y = x \cap y'$ where y' is the complement of y. The complement of a set interval is characterized only by the fact that it does not contain the elements in the lower bound (e.g. c in this case). So the convex closure of a set interval difference is:

$\overline{[a,b] \setminus [c,d]} = [a \setminus d, b \setminus c]$

17.3.2 Set Constraints and Graduations

Primitive set constraints apply to set variables or ground sets. They constrain at most two set variables or a set variable and an integer (for graduated constraints). They can be of the form $S \in [a,b], S \subseteq S_1, S = S_1 \cup S_2, S = S_1 \cap S_2, S = S_1 \setminus S_2, e \in S, e \notin S, |S| \geq c, |S| \leq c$.

Many more constraints can be specified but will be rewritten in term of the primitive ones. For instance, n-ary constraints of the form $s_1 \cup s_2 \subseteq s_3 \cap s_4$. Th reason is that the partial solving of constraints requires us to express each set variable in terms of the others. Since there is no inverse operation for $\cup, \cap, \setminus$ there is no way to move all the operation symbols on one side of the constraint relation. So it is necessary to decompose n-ary constraints into primitive ones unless some global reasoning is sought with dedicated propagators (see next section). The decomposition approach is similar to the relational form of arithmetic constraints over real intervals [18].

To increase the expressiveness of a set solver, and in particular to be able to deal with optimization functions, we apply graduation functions to sets. A graduation maps a no quantifiable term to an integer value denoting a measure of the term. The set cardinality is one example of such a function. Another one is the weight function that sums the element values of the set. Both can then be restricted by arithmetic constraints. The following definitions give necessary conditions to consider graduations for a given set.

Definition 17.3. *A set S provided with an order relation $\preceq$ is graduated if there exists a function f from S to $\mathcal{Z}$ (positive and negative integers) which satisfies:*
$x \prec y \Rightarrow f(x) < f(y)$ *($\prec$ is a strict ordering, $<$ the arithmetic inequality)*
x precedes $y \Rightarrow f(x) = f(y) + 1$
An element x_i precedes an element x_{i+1} if in the chain of elements $x = x_0 \prec x_1 \prec ... \prec x_n = y$ in S there is no other element between them.
f is the graduation of S.

The existence of a graduation of a set which does not correspond to a chain (e.g. a set of set intervals) is guaranteed for the closed set intervals under set inclusion [40]. Furthermore, if there exists one such graduation of a set, then there exists an infinite number of graduations of this set. The weight function is a case in point.

Definition 17.4. *A graduation f is a function from $[\mathcal{D}_S, \subseteq]$ to $\mathcal{Z}$ (set of positive and negative integers) which maps each element $x \in D_S$ to a unique m such that $f(x) = m$.*

The convex closure of a graduation f is required to deal with elements from ΩD_S. The closure function, written $\overline{f}$, maps elements from ΩD_S to a subset of the powerset $\mathcal{P}(\mathcal{Z})$ containing intervals of positive and negative integers. This subset is designated by $\Omega\mathcal{Z}$.

Example 17.5. Let s be a set and $|s|$ its cardinality (a positive integer). Consider the constraint $s \in [\{\}, \{1, 2\}]$. The cardinality function is approximated by $\overline{|\,|}$. Intuitively we have $\overline{|\,|}(s) = [0, 2]$.

Definition 17.6. *Let $f : D_S \rightarrow \mathcal{Z}$. The function $\overline{f} : \Omega D_S \rightarrow \Omega\mathcal{Z}$ is derived from f as follows:*
$\overline{f}([a, b]) = [f(a), f(b)]$

Property 17.7. *If $x \in [a, b]$ then $f(x) \in \overline{f}([a, b])$.*

This property guarantees that the output of the function $\overline{f}$ applied to a set domain contains the actual graduation value of the concerned set variable.

17.3.3 Local Consistency

Local consistency for the primitive constraints individually ensure that the set interval calculus holds. This can be captured in the following definition of bound consistency for constraints over combined domains [13].

Definition 17.8. *A constraint is Bound Consistent (denoted BC), iff for each set (holds also for multiset domain variables), its $lub(s)$ (respectively $glb(s)$) is the union (respectively intersection) of all the values for s that belong to a valid assignment, and for each integer variable x there is a valid assignment that satisfies the constraint for the max and min values in the domain of x. An assignment is valid if the value given to each set (or multiset) is within its domain bounds, and the value given to each integer variable is between the min and max in its domain.*

For the sole case of set and multiset variables, BC can be defined using the characteristic function for each set variable (or occurrence representation for multiset variables). A set constraint is BC if its characteristic function is bounds consistent in the common finite domain terminology [93].

17.3.4 Enforcing BC

The consistency notion defines conditions to be satisfied by set domain bounds, and integer domains so that a set constraint is BC. If such conditions are not satisfied this means that elements in the domain are irrelevant. BC can be inferred by moving such elements "out of the boundaries of the domain" which means pruning the bounds of the domain. The essential point is that a refinement of both bounds allows us to prune a domain. Reducing the set of possible values a set could take can be achieved either by extending the collection of *definite* elements of a set *i.e.*, adding elements to the glb of a set domain, or by reducing the collection of *possible* elements *i.e.*, removing elements from the lub of a set domain. Both computations are deterministic. The inference rules are presented as deterministic rewrite rules that operate when the conditions are met:

$$\frac{\text{conditions}}{\text{constraint store changes}}$$

For set constraints

Consider the constraint $s \subseteq s_1$ such that $s \in [a,b], s_1 \in [c,d]$. Inferring its local consistency amounts to possibly extending the lower bound of the domain of s_2 and to possibly reducing the upper bound of the domain of s_1. This is depicted by the following inference rule:

$$\text{I1.}\ \frac{b' = b \cap d\ , \quad c' = c \cup a}{\{s \in [a,b],\ s_1 \in [c,d],\ s \subseteq s_1\} \longmapsto \{s \in [a,b']\ , s_1 \in [c',d],\ s \subseteq s_1\}}$$

When s, s_1 denote set expressions, the relational forms are created and the following additional inference rule is necessary to deal with the projection functions. For each projection function $\overline{\rho_i}$ describing the domain of an s_i appearing in a set expression, we have:

$$\text{I2.}\ \frac{a'_i = a_i \cup c\ , \quad b'_i = b_i \cap d}{\{\ s_i \in [a_i,b_i],\ \overline{\rho_i} = [c,d]\ \} \longmapsto \{\ s_i \in [a'_i,b'_i]\}}$$

For primitive graduated constraints

The constraint $f(s) \in [m,n]$ such that $s \in [a,b]$ describes a mapping from an element belonging to a partially ordered set to an element belonging to a totally ordered set. Consequently, it might occur that two distinct elements in $[a,b]$ have the same valuation in $[m,n]$. This implies that inferring the local consistency of this constraint might require refining $[a,b]$ only if a single element in $[a,b]$ satisfies the constraint. If this element exists, it corresponds necessarily to one of the domain bounds since they are uniquely defined and are strict subset (or superset), of any element in the domain. Thus, the value of the graded function mapped onto them cannot be shared. The inference mechanism is depicted by the following rules. $min()$ and $max()$ are functions which take as input a collection of integers and return respectively the minimal and maximal integer value of this collection.

$$\text{I3.}\ \frac{[m',n'] = [max(m,f(a)), min(n,f(b))]}{\{\ s \in [a,b]\ , f(s) \in [m,n]\} \longmapsto \{s \in [a,b]\ , f(s) \in [m',n']\ \}}$$

I4. $$\frac{n = f(a)}{\{\, s \in [a,b]\,, f(s) \in [m,n]\} \;\longmapsto \{s = a\,\}}$$

I5. $$\frac{m = f(b)}{\{\, s \in [a,b]\,, f(s) \in [m,n]\} \;\longmapsto \{s = b\,\}}$$

By their definition, the inference rules are correct (all possible solutions are kept), contracting (final domains are subset of the initial domains), idempotent (the smallest domains have been computed the first time) and inclusion monotone (smaller initial domains yield smaller final domains). The consistency of a system of constraints results from the consistency of each constraint appearing in it. A generic algorithm is used to call the relevant inference rules dedicated to enforcing BC. It reduces the set bounds until a fixed point is reached. In the case of set intervals, the algorithm resembles the relaxation algorithm used by CLP(Intervals) systems [62] commonly referred to as fixed point algorithm [11], see Chapter 16, "Continuous and interval constraints".

17.3.5 Illustrative Model

We illustrate a 0-1 model versus a subset-bound set model of a simple bin packing problem [39]. Bin packing problems belong to the class of set partitioning problems. A multiset of n integers is given $\{w_1, ..., w_n\}$ and specifies the weight elements to partition. Another integer W_{max} is given and represents the weight capacity. The aim is to find a partition of the n integers into a minimal number of m bins (or sets) $\{s_1, .., s_k\}$ such that in each bin the sum of all integers does not exceed W_{max}. This problem is usually stated in terms of arithmetic constraints over 0-1 variables and solved using MIP techniques or finite domain constraint programming. It requires one matrix (a_{ij}) to represent the elements of each set, one vector x_j to represent the selected subsets s_k and one vector w_i to represent the weights of the elements a_{ij}. The set model uses a `weight` graded constraint that sums the weights of the items in a set domain.

IP abstract formulation	set abstract formulation
$\sum_{j=1}^{m} a_{ij}\, x_j = 1\ \forall i \in \{1, .., n\}$	$s_1 \cap s_2 = \{\},\ s_1 \cap s_3 = \{\},\ .., s_{n-1} \cap s_m = \{\}$
	$s_1 \cup s_2 \cup ... \cup s_m = \{(1, w_1), .., (n, w_n)\}$
where:	
$x_j = 0..1$ (1 if $s_j \in \{s_1, .., s_k\}$)	$s_j \in [\{\}, \{(1, w_1), .., (n, w_n)\}]$
$a_{ij} = 0..1$ (1 if $i \in s_j$)	$\texttt{weight}(i, w_i) = w_i$;
$\sum_{i=1}^{n} a_{ij}\, w_i \leq W_{max}\ \forall j \in \{1, ..., m\}$	$\forall s_j, \sum_{i=1}^{\#glb(s_j)} \texttt{weight}(i, w_i) \leq W_{max}$

Under these assumptions, the program to solve is to minimize the number of bins:

$\texttt{min}\, x_0 = \sum_{j=1}^{m} x_j$	$\texttt{min} x_0 = \#\{s_j \mid s_j \neq \{\}\}$

17.3.6 Multiset Domains

Multisets can also be used to model the bin packing problem by considering essentially the weights and not the items. They were introduced in the previous section in the context of constructed sets. We present here approaches towards introducing multiset objects that are specified using domains, as they can be naturally seen as extensions to set domains where

the occurrence needs to be taken into account. Multiset domains are not present in many languages yet, but can be found in SOLVER under the name *bags* [50]. As described in [54], the main difference between set and multiset domains in a Constraint Satisfaction Problem sense lies in the maintenance of the occurrence functions. And in fact multisets can be solely defined by means of the occurrence function. Let $occ(m, s)$ be the number of occurrences of m in the multiset s. Multiset operations such as union, intersection, difference, etc, are defined by properties of the occurrence function. We have:

$occ(m, s_1 \cup s_2) = max(occ(m, s_1), occ(m, s_2))$
$occ(m, s_1 \cap s_2) = min(occ(m, s_1), occ(m, s_2))$
$occ(m, s_1 \setminus s_2) = max(0, occ(m, s_1) - occ(m, s_2))$
$s_1 = s_2$ iff $\forall m, occ(m, s_1) = occ(m, s_2)$
$s_1 \subseteq s_2$ iff $\forall m,\ occ(m, s_1) \leq occ(m, s_2)$

Just like sets, different representations are possible for multiset domains. The subset bound representation can be generalized to sets allowing multiple occurrence of elements, and the characteristic function can be generalized to the occurrence vector. Also the list of finite domain variables commonly used to represent sets in Finite Domain (FD) solvers can be used for multisets with the difference that the variables are not constrained to be distinct but each element should appear in a number of variables describing its occurrence. We can compare the expressiveness of the different representations in terms of the multiset values it represents. For instance, the occurrence representation is more expressive than the bound representation (see proofs in [93]). The FD list, also referred to as cardinality representation, is incomparable to either.

Example 17.9. A multiset m_{s1} with possible values $\{\{1, 1, 2\}\}, \{\{2, 2, 2\}\}$can be represented by the "occurrence" vector of integer variables $[x_1, x_2]$ with:

$x_1 =$ occ$(1, ms) \in 0..2$, and $x_2 =$ occ$(2, ms) \in 1..3$
The bound representation for this multiset domain is specified by:
$m_{s1} \in [\{1, 1, 2\}..\{2, 2, 2\}]$
The FD list representation for the same multiset variable is specified by:
$[y_1, y_2, y_3],\ y_1 \in 1..2,\ y_2 \in 1..2,\ y_3 = 2$

Enforcing BC is done by applying inference rules similar to the ones for set constraints taking into account the semantics of the occurrence element (see [93]).

17.4 Influential Extensions to Subset Bound Solvers

`Conjunto` and its peers provide a natural and concise modeling facility for set-based CSPs, space efficient in the representation of large domains, integrated with finite domain solvers through graded function constraints. However, the growing use of such solvers has raised some important shortfalls over the past years, the main ones being the loose approximation of the subset bounds when the actual domains are *sparse*, and the passive use of the cardinality information, ubiquitous in set-based combinatorial problems, and the breaking of problem symmetries. We present the most influential approaches towards improving finite set solvers.

So far, there has been four research directions to strengthen constraint propagation of the first subset bound solvers, built upon `Conjunto` inference rules. These comprise (1) additional cardinality inferences to enrich a subset bound solver; (2) a hybrid set domain that complements the conventional subset domain with lexicographic bounds; (3) a set solver based on a full domain representation using Reduced Ordered Binary Decision Diagrams (ROBDD); (4) global constraint propagators over subset bounds. This section surveys the four of them.

17.4.1 Cardinal

The `Cardinal` solver [3] is a finite set solver in the `Conjunto` style (i.e. subset bound solver) with enhancements to strengthen the use of the cardinality information. `Conjunto` uses the cardinality (and other graded functions like weight) in a unidirectional way, meaning that when a set domain gets refined its cardinality is pruned. The possible inferences from the cardinality to the set have not been considered, mainly due to the practical objective of the language then to remain cost effective in addressing large set-based CSPs (bin-packing, partitioning). However, finite set solvers have a wider applicability. In particular Azevedo applies subset bound solvers to tackle digital circuit diagnosis [4, 3]. For such problems active use of the cardinality information is essential.

Conventional Boolean representations of digital signals consider a pair: a set of faults on which the signal depends, and a Boolean value that the signal takes if there were no faults at all. Both are variables. For instance, $X = \{\{f/0, g/0\}, \{i/1\}\} - 0$ means that signal X is normally 0 but if both gates f and g are stuck-at-0 or gate i is stuck-at-1, then its actual value is 1. Thus $\emptyset$-N represents a signal with constant value N, independent of any fault [4]. The idea of using sets to represent digital circuits is to join the two domains in one by using a transformation, based on a single set domain that approximates both with minimal loss of information. A set representing a pair S-0 is simply represented by the set S, while the pair S-1 is represented by the set $\overline{S}$. The set S can have values $\emptyset$ or D (known set) and is thus given a set interval domain $[0, D]$ whose corresponding cardinality should ideally have only the two possible values $\{0, |D|\}$. Such disjunctive cardinality domains, mapped to sparse set domains, makes the subset bound approximation very loose and ineffective.

New inferences rules are added to the solver to strengthen constraint propagation over the cardinality information, and would benefit such combinatorial problems in particular [4]. Additional cardinality inferences are associated with each basic set operation. To illustrate the pruning power of `Cardinal`, we consider the set difference operation. The following example shows the benefits of additional inference rules using the cardinality information:

Example 17.10. Let s_1, s_2 and s_3 be three set variables such that we have the following system of constraints:

$$s_1, s_2 \subseteq \{a, b, c, d\}, |s_1| = 2, s_3 = s_2 \setminus s_1$$

While traditional subset bound solvers do not infer any information, the `Cardinal` system would infer that $|s_3| \leq 2$. Then any further constraint upon the cardinality of s_3 such as $|s_3| = 3$ would lead to a failure.

The inference rules defined to achieve bounds consistency for the cardinality variables c_1, c_2, c_3 amount to adding new constraints on the cardinality variables to the constraint store. In the case of the set difference constraint we have:

$$c_3 \geq c_1 - c_2$$
$$c_3 \leq c_1 - |glb(s_1) \cap glb(s_2)|$$
$$c_3 \leq |lub(s_1) \cup lub(s_2)| - c_2$$

Note that the `Cardinal` solver first infers arc consistency over the cardinality bounds, at constraint set up which can be useful when cardinality domains are disjunctive like in digital circuits models, but costly in the general case. Thus it maintains bounds consistency over these bounds to remain effective while strengthening constraint propagation. For each primitive set constraints as the set difference above, an inference rule leading to AC for the cardinality domains is first applied.

$$c_3 \in \{n \mid \exists\, i \in D_1, j \in D_2, \max(i-j, i-|lub(s_1) \cap lub(s_2)|) \leq n \leq \min(i-|glb(s_1) \cap glb(s_2)|, |lub(s_1) \cup lub(s_2)| - j)\}$$

Example 17.11 ([3])**.** Consider two sets s_1 and s_2 that can only be $\emptyset$ or $\{f, g, h, i\}$ (i.e. cardinality 0 or 4). To find the initial cardinality domain of their difference $s_3 = s_1 \setminus s_2$, we examine cardinality pairs $\langle 0, 0\rangle, \langle 0, 4\rangle, \langle 4, 0\rangle, \langle 4, 4\rangle$ and conclude that the set difference cardinality is also the pair $\langle 0, 4\rangle$.

The `Cardinal` solver has been implemented atop ECL^{i}PS^e [86] and is fully described in [3, 5]. This solver has shown how finite set solvers can be competitive on problems which were the realm of Boolean algebra. It has demonstrated the expressiveness of finite sets and their applicability to digital circuit design in particular.

17.4.2 Lexicographic Bounds

The ubiquity of the set cardinality information goes beyond digital circuit design and encompasses the large class of combinatorial design problems (e.g. see [19] for a survey) for which set-based CSP models are ideally suited. Examples are sport scheduling, Steiner systems, error-correcting codes. Traditional subset bound solvers have difficulty with such problems as they do not make strong use of the set cardinality information. `Cardinal` offers more in terms of cardinality inferences but such inferences do not propagate onto the subset bounds except for instantiation. This issue is addressed in [84], by extending the domain representation to more closely approximate the true domain of a set variable. This is a complementary approach to `Cardinal` that strengthens the propagation of finite set constraints in a tractable way.

The idea is to consider a set domain ordering that better exploits the cardinality information, and that is also effective at breaking symmetries (when using symmetry breaking constraints) [85]. The new bound representation for set domains is based on an ordering different from the set inclusion (subset order). It is a lexicographic ordering with *lexicographic bounds* specified by $\langle \inf, \sup \rangle$. This ordering relation defines a *total* order on sets of natural numbers, in contrast to the *partial* order $\subseteq$. We use the symbols $\preceq$ (and $\prec$) to denote a total strict (respectively non-strict) lexicographic order.

Definition 17.12. *Let $\preceq$ be a total order on sets of integers defined as follows:*

$$s_1 \preceq s_2 \textbf{ iff } s_1 = \emptyset \vee m_1 < m_2 \vee \big(m_1 = m_2 \wedge s_1 \setminus \{M_1\} \preceq s_2 \setminus \{m_2\}\big)$$
$$\textit{where } m_1 = max(s_1) \textit{ and } m_2 = max(s_2)$$

Example 17.13. Consider the sets $\{1,2,3\}, \{1,3,4\}, \{1,2\}, \{3\}$, the list that orders these sets w.r.t. $\preceq$ is $[\{1,2\}, \{3\}, \{1,2,3\}, \{1,3,4\}]$.

A common use of this ordering is in search problems to break symmetries (e.g. [21] on SAT clauses or [33, 36] on vectors of FD variables). However, this is not the use to which this ordering is put here. It is used on *ground* sets as a means to approximate the domain of a finite set variable by upper and lower bounds w.r.t. this order.

A lex bound domain overcomes one major weakness of the subset bounds, in that the lex bounds denote possible solution sets that satisfy the cardinality restrictions imposed on the set variable.

Example 17.14. Consider a variable X ranging over a subset domain $[\{1,2\}, \{1,2,3,4\}]$, such that X is of size 3. The subset bounds are not a possible instance for X as the domain cannot be pruned to satisfy the cardinality restriction. The lexicographic bounds on the other hand are $[\{1,2,3\}, \{1,2,4\}]$, denoting the min and max sets of size 3 (w.r.t. to the ordering) containing $\{1,2\}$.

Despite its success allowing cardinality constraint to filter the domain more actively, the lex bound representation is unable to always represent certain critical constraints. Primary amongst these constraints is the inclusion or exclusion of a single element. Such constraints are not always representable in the domain because the lex bounds represent possible set instances and not definite and potential elements of a set. In the example above there are sets in between the lex bounds that do not contain $\{1,2\}$, such as $\{4,1\}$. It is the inability to capture such fundamental constraints efficiently in the domain which lead to a hybrid domain of both subset and lexicographic bounds.

The lexicographic ordering for sets is not the only possible definition, nor is it, perhaps, the most common when talking about sets. Its use comes from two reasons: 1) for sets of cardinality 1 it is equivalent to the $\leq$ ordering of FD variables and 2) usefully, it extends the $\subseteq$ ordering and we have:

Theorem 17.15. *[84]* $\forall s_1, s_2 \in \mathcal{P}(U) : s_1 \subseteq s_2 \Rightarrow s_1 \preceq s_2$

Theorem 17.15 is used in the hybrid domain to make inferences between the two bounds representations for set variables.

A collection of inference rules have been defined to propagate primitive set constraints with respect to the lex bounds, subset bounds and cardinality bounds. A prototype hybrid solver has been implemented in ECL^{i}PS^e atop the `ic_sets` library. First results showed spectacular improvements over traditional subset bound solvers, on the network design SONET problem [85] and more pruning but at a substantial computational cost on some combinatorial design problems such as the Steiner triple problems and binary error correcting codes. The main novelty of the approach is the introduction of a new domain representation whose bounds account for the cardinality restrictions and can be used for effective symmetry breaking (using symmetry breaking constraints).

17.4.3 ROBDDs

The problem of efficient finite set reasoning in a constraint logic programming context can also be addressed from a radically different perspective as described in [60]. The idea was first motivated by rejecting the belief that the very large number of values of a finite set domain precludes a precise and un-approximated representation, and instead to show how Reduced Order Binary Decision Diagrams (ROBDDs) can be used to represent full set domains and set constraints in a compact manner. Using existing efficient libraries to represent and manipulate these compact data structures, Lagoon and Stuckey demonstrate techniques for combining ROBDDs in ways that correspond to basic finite set constraints (e.g. $\setminus$, $\cap$, $\cup$, $||$) which minimize the size of the resulting ROBDD [60]. An ROBDD is a *canonical function representation (up to reordering)* of a Binary Decision Diagram which permits and efficient implementation of many Boolean function operations [16].

Let s be a set variable, and let $\{1, .., N\}$ be its domain of possible values. The ROBDD domain representation makes use of the characteristic function that defines the one-to-one correspondence between a subset s of a known set S and a Boolean algebra:

$$f : x_i \rightarrow \{0, 1\} \text{ such that } f(x_i) = 1 \text{ iff } i \in s$$

Hence a set variable s is represented by a vector of Boolean variables $< x_1, .., x_N >$. Now if we consider an assignment A of values to variables, each x_i will take value one if and only if $i \in s$. The i's are first drawn from a universe of discourse. Such an assignment can be represented as a Boolean formula $B(A)$:

$$B(A) = \bigwedge_{i \in U} y_i \text{ where } y_i = \begin{cases} x_i & \text{if } i \in A \\ \neg x_i & \text{otherwise} \end{cases}$$

Each known set can be seen as an assignment, hence the full domain of a set variable $D(s)$ can itself be represented by a Boolean formula $B(D(s))$. This formula is a disjunction of $B(A)$ over all possible sets A in $D(s)$ [45]:

$$B(D(s) = \bigvee_{A \in D(s)} B(A) \text{ where } B(A) \text{is defined above}$$

Example 17.16. Let $U = \{1, 2, 3\}$ and let s be a set variable with $D(s) = \{\{1\}, \{1,3\}, \{2,3\}\}$. We associate Boolean variables $\{v_1, v_2, v_3\}$ with s given U. $D(s)$ is the Boolean formula $(v_1 \wedge \neg v_1 \wedge \neg v_3) \vee (v_1 \wedge \neg v_2 \wedge v_3) \vee (\neg v_1 \wedge v_2 \wedge v_3)$. The three solutions to this formula correspond to the elements of D(s).

While such a formula can be constructed using an ROBDD, in practice the approach only ever constructs the ROBDD for a domain implicitly through constraint propagation. The ROBDDs are used to model the constraint themselves. Indeed any set constraint can be converted to a Boolean formula.

Example 17.17. Let $U = \{1, 2, 3\}$, and the constraint $s_1 \subseteq s_2$. Assume that the Boolean variables associated with s_1 and s_2 respectively are v_1, v_2, v_3 and w_1, w_2, w_3. The inclusion constraint can be represented by the Boolean formula: $(v_1 \rightarrow w_1) \wedge (v_2 \rightarrow w_2) \wedge (v_3 \rightarrow w_3)$. This formula can be represented by two different ROBDDs depending on the variable ordering.

ROBDDs are ordered and thus require an ordering of the Boolean variables used. The order can have a drastic effect on the size of the ROBDDs when constraints are represented, i.e. when there is a specific relationship between elements of the universe.

The Boolean approach allows the ROBDD-based modeling to be extended to handling integer and multiset constraint problems as well as some global set constraints (comprehensive description in [45]). While initially motivated by using a full set domain representation that do not approximate the possible set values, ROBDD have also been used to model less strict consistency notions and domain approximations, such as set bounds, cardinality bounds and lexicographic bounds consistency; with a thorough comparative evaluation of the different domain representations [44, 45].

The ROBDD-based solver offers a flexible modelling facility and has shown high performance results on several standard combinatorial design constraint problems. However, it does require the use of Boolean formula and variables to model such problems.

17.4.4 Global Set Constraints

The above works strengthen constraint propagation in complementary ways by revising the concept of set domain or enriching the local inference rules. A more traditional approach in constraint programming to offer a better tradeoff "natural formulation"/efficiency consists in deriving global propagators for a class of symbolic constraints, see Chapter 7, on Global Constraints. This was not considered in finite set solvers till recently, at least in published academic articles, but is now contributing interesting results.

Global reasoning on a class of symbolic set constraints, first considered some n-ary constraints like the `atmost1` (sets intersecting pairwise in atmost one element), or its complement, the `distinct` constraint (sets that differ pairwise in atleast one element) over sets of fixed cardinality [82]. Such constraints and other n-ary constraints like `union` and `disjoint` have been used in set-based constraint languages but essentially as syntactic abstractions of collections of binary or ternary constraints, solved with local consistency techniques. The ubiquity of set intersection in conjunction with cardinality restrictions in set-based combinatorial problems drove the research agenda towards more efficient propagators.

Example 17.18.
$[s_1, s_2, s_3] \in \{\{\}..\{a, b, c, d\}\}$
$|s_1| = |s_2| = |s_3| = 2$
`disjoint`$(\{s_1, s_2, s_3\}$

BC on this system of constraints does not detect inconsistency. However, if the cardinality constraints are combined with the disjointness constraint one can see that there are no solutions by doing a simple pigeon hole test. This can be deduced if we consider the set of constraints globally. In fact, the representation of sets within powersets specified as set intervals can be used to derive some global inferences based on combinatorial analysis formulas. A simple satisfiability test can first be checked (ie. pigeon hole test), determining whether a set of 4 elements can be partitioned into 3 sets of 2 which fails ($\frac{4}{3} \neq 2$). A more elaborate test that does not require the sets to have same cardinalities derives an upper bound on the number of possible partitions of 4 elements into 3 sets of cardinality 2. Such numbers are known as a Stirling number $\frac{4!}{(2!)^3(3!)} = \frac{1}{2}$ [12]. If it is less than one, the

problem is unsatisfiable since there isn't a single possible partition. However if the number is greater than one we would know how many different partitions there are.

There exist some counting functions that determine the maximum number of configurations allowed in a superset S, given some shared properties, see [82]. When considering the values of these functions on can then investigate how and when they can be used effectively, first to detect unsatisfiability but also to prune further irrelevant set values in an a priori manner. The counting functions provide a mathematical information that is not easily deducible in logic. They enable the definition of a set of inference rules to strengthen propagation on global constraints such as atmost1 , distinct over fixed cardinality sets. Such rules do not infer BC but are tractable.

Decomposition and complexity

The problematic of deriving inference rules without a clear idea of how much we do or can infer, and how far we can go towards global reasoning raises fundamental theoretical issues. This lead to a systematic study of several aspects of global constraints and global set and multiset constraints in particular. The approach determined whether decomposition hinders Bounds Consistency (BC), and when it does whether there exists a polynomial algorithm to infer BC on the considered global constraint [93, 13].

For instance, BC on the n-ary `disjoint` is equivalent to BC on its decomposition into binary constraints (pairwise empty intersection). Basically, this holds because any set can be assigned the empty set. However, when the set cardinalities are constrained (and not zero), –which is frequent in combinatorial design problems for example– the equivalence no longer holds. It was also proved that decomposition of the `atmost1` constraint hinders propagation and that enforcing BC on this constraint is NP-hard.

We summarize the complexity results in Table 17.1. Decomposable implies polynomial, since existing algorithms to infer BC on a set of binary or ternary set constraints are indeed polynomial. Results hold for both set and multiset domains unless specified otherwise. The acronyms stand for: NE (non empty), FC (fixed cardinality). The constraints are classified in terms of the intersection constraints and cardinality restrictions involved. For example the `atmost1` constraint corresponds to pairwise intersect in at most one element ($k = 1$) for fixed cardinality sets, which is the second column of first table.

If now we add the union constraint ($\bigcup_i s_i = s$) to the intersection ones we obtain covering problems. The first left column above becomes a partition constraint for which results are known. The other columns are yet open problems.

$|s_i \cap s_j| = 0$ Partition is decomposable and *polynomial*
+ $\forall k,\ |s_k| > 0$, NEpartition is not decomposable but *polynomial*
+ $\forall k,\ |s_k| = c_k$, FCpartition also referred to as `partition`
is not decomposable and is *polynomial on sets, NP-hard on multisets*

It is important to note that the global constraints applied to multisets versus sets diverge on the two most important constraints (from an application point of view): fixed cardinality disjoint and partition constraints. We describe below the existing algorithms to infer BC when the constraints apply to sets, however doing so over multisets has been proved to be NP-hard [13].

Table 17.1: Summary of complexity results (based on [13]).

$\forall k...$	$\lvert s_i \cap s_j \rvert = 0$	$\lvert s_i \cap s_j \rvert \leq k$	$\lvert s_i \cap s_j \rvert \geq k$	$\lvert s_i \cap s_j \rvert = k$
-	Disjoint decomposable *polynomial*	Intersect$_{\leq k}$ decomposable *polynomial*	Intersect$_{\geq k}$ decomposable *polynomial*	Intersect$_{=k}$ not decomp. *NP-hard*
$\lvert s_k \rvert > 0$	NEdisjoint not decomposable *polynomial*	NEintersect$_{\leq k}$ decomposable *polynomial*	NEintersect$_{\geq k}$ decomposable *polynomial*	NEintersect$_{=k}$ not decomp. *NP-hard*
$\lvert s_k \rvert = c_k$	FCdisjoint `disjoint` not decomposable *polynomial on sets* *NP-hard on multisets*	FCintersect$_{\leq k}$ `atmost1` not decomposable *NP-hard*	FCintersect$_{\geq k}$ not decomposable *NP-hard*	FCintersect$_{=k}$ not decomp. *NP-hard*

Algorithms for the `disjoint` and `partition` constraints

The basic case of disjoint and partition is decomposable for the reasons we gave above. However, when sets have fixed cardinality, decomposition of these constraints hinders constraint propagation and thus deriving a global propagator is necessary to ensure BC. We describe how these constraints have been solved in the literature. Two lines of work have been undertaken to derive similar algorithms for the global `disjoint` and `partition`.

Based upon counting functions from design theory, the first approach derived four global conditions which must hold for disjoint sets of fixed cardinality [83]. Using an extension of Hall's theorem [43], the authors proved that these conditions, if satisfied, were sufficient to ensure BC. The actual proof procedure constitutes the basis of the algorithm which actually corresponds to an augmenting network in a max-flow problem, and is similar to a combination of a flow/matching and a Strongly Connected Component (SCC) algorithm (see [83]). Interestingly this implementation corresponds closely to the GAC algorithm for the Global Cardinality Constraint (GCC) [79], see chapter 7, "Global Constraints", and we show the reasons why below. This algorithm also holds for the `partition` constraint since the only pruning achievable on the disjoint is when one can identify minimal partitions within the constraint (i.e strongly connected components). So one needs to identify partitions in order to do any global pruning on the disjoint constraint.

Using the GCC constraint. The GCC constraint applies to a family of finite domain variables with set of values in B. It constrains the number of times (cardinality) an element of B can be assigned among the different variables.

The use of the GCC constraint to resolve the `disjoint` and `partition` constraints is offered in ILOG solver and Configurator [50, 51] and has been recently described in [13].

The main idea is to formulate each of the two global set constraint with a *dual FD model* based upon the GCC constraint. The semantics of the disjoint constraint is as follows. Let `disjoint`$(s_1, .., s_i, .., s_m)$ constrains the set variables s_i such that $\forall i \in B, s_i \in A, |s_i| = c$. The disjointness constraint ensures that no element of A (domain of the s_i) is added to two different set variables.

This constraint has an equivalent formulation in the language of finite integer variables where one seeks to assign a set identifier to a FD variable. This formulation is called dual because the initial set variables become values and the set elements become variables. The equivalent dual formulation uses the GCC constraint, as presented below.

Consider the `GCC`$(\{y_1, .., y_j, .., y_n\}, B', C)$ constraint such that $\forall j \in A, y_j \in B'$ (with $B' = B \cup \varepsilon$ with ε being a dummy value for the case where j is unassigned). The global cardinality constraint limits to $C[j] = 1$ in the disjoint case, the number of times an element i from B' is assigned to a variable y_j (ranges between $0..\infty$ for the dummy variable). We have $n = m \times c$ FD variables. The dummy value is necessary since there might be some values in A that don't belong to any set at all. The set model and dual FD model with GCC constraint are equivalent. A solution to the first model can be mapped to a solution of the second model and vice versa by applying the following one-to-one mapping:

$$\text{for } i \in A : y_j = i \textbf{ iff } j \in s_i$$

Thus a solution is consistent with the set model if and only if its dual FD representation is consistent with the GCC model. The complexity of both the GCC and set based algorithm is in $\mathcal{O}(m^2 c)$, with m the number of sets and c their cardinality [83, 13].

Further remarks.

- Note that the equivalence between the two models holds because the constraints represent an injective mapping from a set of elements into a set of sets (each element belongs to at most one set). As soon as an element can belong to more than one set we have a surjective mapping and the dual approach based on bipartite graph, and network flow model would not apply.

- This dual approach also holds for the `partition` constraints over fixed cardinality sets with the only difference that all elements must be assigned and thus the dummy value is removed.

More recently, the application of existing global constraints over finite domain variables to other domains has been considered. For instance, the `all-different` and `GCC` global constraints have been extended to variables whose values are multisets, sets or tuples [78]. Note that a tuple is represented as a list of finite domain variables as opposed to having a tuple domain with tuples as elements. The issue for such domains is the large domain size. A binomial representation is proposed to address this aspect. Existing global propagators are used in combination with efficient enumeration algorithms for large domains.

17.5 Constraints over Maps, Relations and Graphs

17.5.1 ALICE Legacy

As mentioned earlier, the seminal work of Laurière was motivated by a need to *state combinatorial problems simply by constraining relation and graph objects over finite sets* [61]. It aimed at clearly separating the problem statement from its solving. The motivation was to allow a combinatorial search problem to be formulated in the most concise and natural manner.

The constrained object was not associated with a domain (set of values the relation can take) and was not "pruned" using consistency techniques, rather it was mapped to an internal representation based on a bipartite graph structure. Operations were performed on this structure.

In ALICE, constraints are expressed in a mathematical language based on relation theory and some notions of graph theory. The searched objects are functions which should satisfy a set of constraints. The solver combines a depth-first search method with sophisticated constraint manipulation techniques and a set of powerful heuristics. The lack of flexibility of this seminal system both in the language representation and the solving strategy motivated the design and implementation of CHIP.

It has also motivated numerous works in the development of high level specification languages for combinatorial problems. Such proposals have been revived in the past years and we can now see two clear trends in the design of high level constraint languages over maps, relations and graphs objects:

- a class of *programming languages* over new constraint domains, where functions relations or graphs become constrained objects. The resolution algorithms depend then upon the representation of the new constrained objects. Most of these works are still novel and currently mapped down to finite set solvers as we will see below.

- a class of *modeling languages* offering high level constructs such as functions, maps and sequences to model combinatorial problems in a concise manner. Such approaches do not reason directly about the constrained object to solve the specified problem. Instead, the formulation is compiled into a lower language benefiting usually from existing solvers.

17.5.2 Constraint Programming Beyond Sets

The extension of constraint solvers with high modelling and programming facilities has lead to the definition of new constraint domains over binary relations, graph and maps essentially. We will present their main components.

Relation variables. When dealing with sets, it sounds quite natural to deal with relations as well. The `Conjunto` language —mainly designed to handle finite set constraints— also provides relations at the language level to extend the expressive power of the language when dealing for example with circuit problems and matching problems originating from Operations research. Relation terms are basically built using set terms.

A relation $\mathcal{R}$ is commonly represented as a set of ordered pairs (x_i, y_j) such that x_i belongs to the DS-domain *d* of $\mathcal{R}$ and y_j to its AS-range[1] *a*. In other words, a relation $\mathcal{R}$ on two ground sets d and a is a subset of the Cartesian product $d \times a$. Keeping this representation to deal with relations as specific set terms containing pairs of elements can be very costly in memory. Indeed, the statement of the Cartesian product referring to a relation requires us to consider explicitly a huge set of pairs. This is very inconvenient. Instead, a relation in `Conjunto` is represented as a specific data structure which is characterized by two ground sets (DS-domain and AS-range) and a list containing the successor sets attached to each element of DS-domain.

Considering one successor set per element splits the domain of a relation into a collection of set domains. The resulting value of a relation is clearly the union of the successor sets. This approach is close to the one introduced in ALICE which dealt essentially with functions. However, in ALICE there is no explicit notion of set domain.

Definition 17.19. *Let a relation be $r \subseteq d \times a$. The successor set s of an element $x \in d$ is the set $s = \{y \in a \mid (x, y) \in r\}$.*

The definition of constraints applied to relation variables abstracts from stating directly constraints over the set DS-domain and AS-range or over the successor sets. The following injection, map, surjection, bijection constraints over a relation r have been embedded in `Conjunto`. We illustrate some of them below. They are represented using the cardinality operation $||$, the usual set operation symbols ($\cup, \cap$) and the arithmetic inequality ($\geq$).

Constraints	Interpretation
r `bin_r` d `-->` a	r = `birel`(l, d, a) where $l = \{s_i \mid \ \forall i \in d,\ s_i \ \in \{\}..a\}$
(i, j) `in_r` r	if $i \in d, j \in a$ then $j \in s_i$
`funct`(r)	$\forall i \in d,\ \|s_i\| = 1$
`inj`(r)	$\|d\| \leq \|a\|,\ \|d\| = n$ $s_1 \cap s_2 = \emptyset,\ s_1 \cap s_3 = \emptyset,\ ...,\ s_{n-1} \cap s_n = \emptyset$ $\forall i \in d,\ \|s_i\| = 1$
`surj`(r)	$\|d\| \geq \|a\|,\ \|d\| = n$ $s_1 \cup s_2 ... \cup s_n = a$ $\forall i \in d,\ \|s_i\| = 1$

These constraints do not require any specific solver since the reasoning is based on the successor set variables. Such constraints were used to prototype partitioning problems.

Graph and map variables. In the same line of work, the `CP(Graph)` language has been designed to tackle combinatorial problems involving "subgraph findings" common in the fields of communication networks, route planning and more recently bio-informatics [24].

`CP(Graph)` deals more specifically with graphs and graph constraints and represents a graph domain by considering its nodes and arcs, that is a graph $g = (sn, sa)$ is defined by a set of nodes sn and a set of arcs $sa \subseteq sn \times sn$. It handles both directed and undirected graphs and offers a set of kernel constraints used to derive other graph constraints.

Similarly to subset bound solvers, `CP(Graph)` builds upon a partial ordering among graphs to reason upon graph domains. We have, given $g_1 = (sn_1, sa_1)$ and $g_2 = (sn_2, sa_2)$:

[1] DS-domain and AS-range stand respectively for departure and arrival sets

$g_1 \subseteq g_2$ iff $sn_1 \subseteq sn_2 \wedge sa_1 \subseteq sa_2$

Graph domains are represented by the lattice of graphs partially ordered by set inclusion and specified by a graph interval $[g_L, g_U]$ such that g_L is the greatest lower bound and g_U the least upper bounds of the lattice. It also considers the arcs and nodes as set variables. The use of additional node and arc variables adds expressiveness to the language when describing complex graph constraints.

Dooms et al. show that any complex graph constraint can be expressed using a combination of the following kernel graph constraints:

- $Arcs(g, sa) \in [sa_L, sa_U]$ where sa describes the set of arcs of g that range over the subset bound domain
- $Nodes(g, sn) \in [sn_L, sn_U]$ where sn describes the set of nodes of g that range over a subset bound domain
- $ArcNode(a, n_1, n_2)$ states that the arc variable a is an arc between two nodes n_1 and n_2

A set of propagation rules allows to infer arc consistency over these constraints. The expressiveness of the constraints allows `CP(Graph)` to define more complex graph constraints based upon the kernel constraints. We illustrate some of them. The functional form of the kernel constraints is used to ease readability.

The $SubGraph(g_1, g_2)$ constraint can be specified by:

$$SubGraph(g_1, g_2) \equiv Nodes(g_1) \subseteq Nodes(g_2) \wedge Arcs(g_1) \subseteq Arcs(g_2)$$

The $InNeighbors(g, n, sn)$ constrains sn to be the nodes in g for which an inward arc incident to n is present.

$$\begin{aligned} InNeighbors(g, n, sn) \equiv sn \subseteq Nodes(g) \wedge (|sn| > 0 \Leftrightarrow n \in Nodes(G) \\ \wedge \forall i \in Nodes(g_U) : n \in sn \Leftrightarrow (i, n) \in Arcs(G) \end{aligned}$$

Clearly even tough the kernel allow us to express any graph constraints, such formulations might not be effective. Thus CP(Graph) also offers global graph constraints based on existing results from literature in the field, see Chapter 7, “Global Constraints”.

Recent advances in CP(Graph) include its extension to manipulate *map* terms as well in CP(Graph + Map)[22]. The main application of CP(Map) is for graph pattern problems. The language extends the relation terms of `Conjunto` (built upon domain and range limited to ground sets) where domain and range become variables. As the departure set of the map is not a ground set, instead of using a list of successor sets, `CP(Map)` uses an indexed array. As maps are functions and not general relations, the domain variables stored in this indexed array are not finite sets but finite domain variables.

17.5.3 High Level Modeling/Specification Languages

Recent proposals have considered Map variables as high level type constructors, simplifying the modeling of combinatorial optimization problems, which would then be compiled into another programming language. We outline recent results in this related area of constraint modeling. The language ASRA defines a relation or map variable from a set v to a

set w, where supersets of v and w must be known [32]. While the map variables and constraints are used to model a constraint problems, the resolution of the model is handled by another system. In this proposal, the derived model are compiled into OPL[91]. This idea can also be found in the language $\mathcal{L}$ where v and w are ground sets[48]. Finally, relation and map variables are also described in [35] as a useful abstraction in constraint modelling. Rules are proposed for refining constraints on these complex variables into constraints on finite integer and set variables.

17.6 Constraints over Lattices and Hierarchical Trees

Proposals for higher computation domains have been made recently which deserve attention. These include the generalization of existing interval based approaches to propose a generic framework for defining and solving interval constraints on any set of domains (finite or infinite) that are lattices [30]. The approach is based on the use of a single form of constraint similar to that of an indexical used by Constraint Logic Programming for finite domains and on a particular generic definition of an interval domain built from an arbitrary lattice. They provide the theoretical foundations for this framework and a schematic procedure for the operational semantics. Examples are provided that illustrate how new (compound) constraint solvers can be constructed from existing solvers using lattice combinators and how different solvers (possibly on distinct domains) can communicate and hence, cooperate in solving a problem.

Another challenging domain is that of order-sorted domains and ontologies. Both proposals are driven by industrial needs. The first one shows how constraint satisfaction techniques can be extended to address order-sorted domains, from class taxonomies with an object oriented perspective [17]. The use of ontologies, is itself motivated by applications for the configuration of product and services, for instance in the e-commerce [59]. This second approach defines a constraint domain where all values that a variable may take are organized into a hierarchy. Such hierarchies are often called ontologies or thesauri in Artificial Intelligence. Both approaches are quite close. The objective is to define a system that would allow the use of order-sorted domains in constraint programming for modeling purposes. The outlined algorithmic approach to reason about ontologies follows the bound and convex interval reasoning of finite set intervals. Other approaches to deal with hierarchies have essentially used the standard CSP formalism and constrain the values of properties as opposed to the entities in a hierarchy itself [34].

17.7 Implementation Aspects

We present some of the core implementation issues mainly relating to subset bound solvers since they are the main practical language implementations and are used by higher level constructs as well. For example the `CP(Graph)` prototype is built over the FD and finite set solver of `OZ`.

17.7.1 Existing Subset Bound Solvers

Subset bound solvers can be found atop different types of kernel languages such as Prolog enriched with constraint solving and replacing the standard Prolog variable by an attributed

variable [52] subject to a dedicated unification algorithm. Prolog based set solvers can be found in ECL^{i}PS^e, `B-Prolog` and `Cardinal` for instance. Other kernel systems are based on object oriented language such as C^{++}(`SOLVER`), concurrent object-oriented language like OZ (`MOZART`), a functional language `OCaml` (`FACILE`), and java (the open source `Choco` system) to name the main ones. Each offers different modeling and resolution facilities.

17.7.2 Set Data Structures

Most existing finite set solvers make use of the subset bound representation for space and computational efficiency reasons. The ROBDD proposal investigates the use of binary decision diagram to represent set domains, allowing for full domains as well as intervals.

The internal representation of sets plays a role in the time complexity of the different set operations on the domains since such operations cannot be considered constant unlike arithmetic operations over integers. For the bound representation we can use 2 sorted lists one for each bound, an array of 0-1 variables (both bounds in a single array) or bitmaps representing the characteristic function of the set. The same structures can be used for ground set representations if the two bounds are stored separately as well as more elaborate ones such as binomial trees or binary trees.

Since set operations on domains are performed by reasoning on either or both bounds we give hereafter the time complexity for basic set operations on ground sets. When one structure is used to embed both bounds the same reasoning applies. Let s be the set with largest domain such that $d = |lub(s)| + |glb(s)|$. The cardinality information is usually maintained dynamically as part of the set variable data structure.

ROBDDs correspond to directed acyclic graphs. Recall that the ROBDD approach transforms set constraints into Boolean operations and can model domain reasoning as well as interval reasoning. The complexity of basic set operations depends on the ordering of the Boolean variables. For a given constraint we can generate an exponential as well as a linear representation in a Boolean formula. We consider N as the size of the set domain which can potentially correspond to $2^{lub(s)}$. The main thing is that each basic set operation generates an ROBDD. So the complexity issue relates to the size of the generated ROBDD.

17.7.3 Complexity of Set Operations

For bound domains the corresponding initial ROBDD corresponds to the size of the lower bound independent of the upper bound size and any update can be represented in O(| $glb(s)$ | $+N-$ | $lub(s)$ |). For an extensive domain representation the size of the initial ROBDD is linear relative to N [60]. The size of the ROBDD for the different basic set operations is given below where N is the size of the largest set domain and k a bound on the cardinality. The cardinality constraint is quite tricky to express in Boolean formula and requires a quadratic number of formula defined recursively hence the complexity results.

The strength of hash tables is the constant time on average to retrieve information. "+" represents the "capacity" of the backing (the number of buckets).

Alternative approaches exist based on the representation of a ground set. They are used mainly for dynamic set operations (add, remove, and sometimes union) and correspond to tree structures (B-tree, binary search tree, binomial tree). The worst case time complexity for ground sets operations is usually measured by the height of the tree. For

sets of cardinality c we have: $h = \log c$ where h is the height of the tree. For such structures the efficiency lies in the membership test $\mathcal{O}(\log c)$, union is in $O(c \log c)$.

	$=$	$\subseteq$	$\cup$	$\cap$	$\setminus$	$\in$	$\|\|$
sorted list	$O(d)$	$O(glb(s))$	$O(d)$	$O(d)$	$O(d)$	$O(\mid glb(s) \mid)$	$O(1)$
0-1 array	(1)	(d)	$O(d)$	$O(d)$	$O(d)$	$O(1)$	$O(1)$
hash table	$O(1)$	$O(k+)$	$O(k+)$	$O(k+)$	$O(k+)$	$O(1)$	$O(1)$
ROBDD	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$	$O(1)$	$O(k(N-k))$

17.8 Applications

Each structured domain was developed to address particular application needs. For instance, the graph domain was motivated by a problem for biochemical network analysis [24]. The order-sorted and ontology domains were driven by industrial problems, for instance in the area of e-commerce for ontologies.

The structured domain which has been the most widely developed and used is certainly that of finite sets. The reason is probably that sets are the underlying structured objects for the other domains. Set solvers have been used to tackle small and large size benchmark and "real-world" problems ranging from bin packing ([39]), set partitioning ([40, 70, 68]), digital circuit and warehouse location [3], combinatorial design ([8, 60, 45]), and network design ([29, 88, 85]) among others. Recently, combinatorial designs have shown to have a wide applicability in error-correcting codes, sport scheduling, Steiner systems and more recently networking and cryptography (e.g. see [19] for a survey). Set constraints have shown their adequacy for such problems, and powerful models have been derived combined with symmetry breaking techniques and heuristic techniques. We draw particular attention, to the solving of the challenging Kirkman school girl problem in few seconds, with an elaborate approach which uses a set model extended with redundant constraints and symmetry breaking techniques[8]. More discussions on symmetry breaking and modeling aspects can be found respectively in chapter 10, "Symmetry in Constraint Programming", and chapter 11 "Modelling".

Another application area of increasing interest for constraint practitioners, is network design. Various successful set-based models have been proposed to tackle the network design SONET problem from a constraint programming perspective [88, 85]. They demonstrate the strength of applying dual models, redundant constraints and symmetry breaking techniques to set models.

17.9 Further Topics

Constraint reasoning over structured domains has mainly been motivated by the development of high level modeling and specification languages that ease the formulation of complex combinatorial problems while retaining efficiency.

Research on high level specification languages has long existed but is now growing in constraint programming [32, 48, 15]. Many constraint programming languages –both in academia and industry– utilizing structured domains have been proposed, demonstrating

important progress (e.g. graphs [24], order-sorted domains [17], ontologies [59], multi-sets [93], and lattices [30]). Much progress has also been made on improving language effectiveness, in particular with respect to set solvers (e.g. cardinality inferences [5], the use of ROBDDs [45], more expressive domain representations [84, 41], global propagators [82, 13, 51]). This research area is extremely active.

Finally, a programming language that allows practitioners to state the problem in a natural and concise form without needing to worry about the solution method does not yet exist. However, certain steps have been taken towards this goal. In particular, a high level problem formulation allows language designers and programmers, to see the actual problem structure and components, and consequently to identify combinations of constraints that best exploit the problem structure.

Acknowledgements

The author is thankful to Jean-François Puget for his comments during the preparation of this chapter. The author was partially supported by the Royal Academy of Engineering, on a Global Research Award.

Bibliography

[1] J-R. Abrial. The B Book - Assigning Programs to Meanings. *Cambridge University Press*, ISBN = 0521496195, 1996.

[2] A. Aiken. Set Constraints: Results, Applications andFuture Directions. In *Proceedings of PPCP'04*, 1994.

[3] F. Azevedo. *Constraint Solving over Multi-Valued Logics. Application to Digital Circuits*. Frontiers in Artificial Intelligence and Applications, 2003.

[4] F. Azevedo and P. Barahona. Cardinal: an extended set solver. in *Proceedings of Computational Logic*, 2000.

[5] F. Azevedo. Cardinal: A Finite Set Constraints Solver. In Constraint journal, (to appear), 2006.

[6] L. Bachmaier, H. Ganzinger, and U. Waldmann. Set Constraints are the Monadic Class. In *Proceedings of LICS*-1993.

[7] N. Barnier and P. Brisset. Facile: A Functional Constraint Library. In *CICLOPS'01* workshop, help alongside with CP-2001.

[8] N. Barnier and P. Brisset. Solving the Kirkman's Schoolgirl Problem in a Few Seconds. In M. Wallace, editor, *Proceedings of CP*-2004.

[9] P. Baptiste, B. Legeard, and H. Zidoum. Sequence Constraint Solving in Constraint Logic Programming. In *ICTAI*-1994.

[10] C. Beeri, S. Naqvi, O. Shmueli, and S. Tsur. Set constructors in a logic database language. In *Journal of Logic Programming*, pages 181–232, 1991.

[11] F. Benhamou. Interval Constraint Logic Programming. In A. Podelski, editor, *Constraint Programming: Basics and Trends*, LNCS 910, 1995.

[12] C. Berge. Principle of combinatorics. Volume 72 of Mathematics in science and engineering. Academic Press, 1971.

[13] C. Bessière, B. Hnich, E. Hébrard, and T. Walsh. Disjoint, Partition and Intersection Constraints for Sets and Multiset Variables. In M. Wallace, editor, *Proceedings of CP*-2004, LNCS 3258.

[14] C. Bessière, B. Hnich, E. Hébrard, and T. Walsh. The Tractability of Global Constraints. In M. Wallace, editor, *Proceedings of CP*-2004, LNCS 3258.

[15] F. Bouquet, B. Legeard, and F. Peureux. CLPS-B: A Constraint Solver to Animate a B Specification. *International Journal on Software Tools for Technology Transfer, STTT*. 6:2, pp 143–157, Springer Verlag, 2004.

[16] R.E. Bryant. *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams*. ACM Comput. Surv., 24(3), 293–318, 1992.

[17] Y. Caseau and J.-F. Puget. Constraints on Order-Sorted Domains. In *ECAI workshop*, 1996.

[18] J.G. Cleary. Logical arithmetic. *In Future Generation Computing Systems*, chapter 2(2),1987.

[19] Colbourn, Dinitz, and Stinson. Applications of Combinatorial Designs to Communications, Cryptography, and Networking. In *Surveys in Combinatorics, London Mathematical Society Lecture Note Series 187*. Cambridge University Press, 1999.

[20] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, bases théoriques et développements actuels. *T.S.I. (Techniques et Sciences Informatiques)*, 2(4),1983.

[21] J. Crawford, M. Ginsberg, E.M. Luks, and A. Roy. Symmetry breaking predicates for search problems. In *Fifth Int. Conf. on Knowledge Rep. and Reasoning*, 1996.

[22] Y. Deville, G. Dooms, S. Zampelli, and P. Dupont. CP(Graph + Map) for Approximate Graph Matching. In *Proceedings of BeyondFD'05, First International Workshop on CP beyond FD*, held alongside CP-2005.

[23] M. Dincbas, H. Simonis, and P. Van Hentenryck et al. The Constraint Logic Programming Language CHIP. In *Proceedings of FGCS*-1988.

[24] G. Dooms, Y. Deville, and P. Dupont. CP(Graph): Introducing a Graph Computation Domain in Constraint Programming. In *Proceedings of CP*-2004.

[25] A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Logic Programming Language with Finite Sets. In *Proceedings of ICLP*-1991.

[26] A. Dovier. *Computable Set Theory and Logic Programming*. PhD Thesis TD-1/96, Universitàdegli Studi di Pisa, dip. di Informatica, March 1996.

[27] A. Dovier, E. G. Omodeo, E. Pontelli, and G. Rossi. {log}: A Language for Programming in Logic with Finite Sets. *In Journal of Logic Programming*, 28(1), 1996.

[28] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and Constraint Logic Programming. *In ACM Transaction on Programming Language and Systems*, 22(5) 2000.

[29] A. Eremin, F. Ajili, and R. Rodosek . A Set-based Approach to the Optimal IGP Weight Setting Problem. In *Proceedings of INOC*-2005.

[30] A.J. Fernandez and P.M. Hill. An Interval Constraint System for Lattice Domains. *in ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1), ACM Press, 2004.

[31] R. E. Fikes. Ref-arf: A system for solving problems stated as procedures. *Artificial Intelligence*, 1:27–120, 1970.

[32] P. Flener, B. Hnich, Z. Kiziltan. Compiling high level type constructors in constraint programming. In *Proceedings of PADL*-2001, LNCS 1990.

[33] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Proceedings of CP*-2002,

LNCS.

[34] D. W. Fowler, D. Sleeman, G. Wills, T. Lyon, and D. Knott The Designers' Workbench: Using Ontologies and Constraints for Configuration. In 24^{th} *International Conference on Innovative Techniques and Applications of AI*, 2004.

[35] A.M. Frisch, C. Jefferson, B.M. Hernandez, and I. Miguel. The Rules of Constraint Modelling. In *Proceedings of IJCAI*-2005.

[36] I.P. Gent, P. Prosser, and B.M. Smith. A 0/1 encoding of the gaclex for pairs of vectors. In *ECAI/W9 Modelling and Solving Problems with Constraints*, 2002.

[37] C. Gervet. New Structures of Symbolic Constraint Objects: Sets and Graphs. In *Third Workshop on Constraint Logic Programming (WCLP'93)*, 1993.

[38] C. Gervet. Sets and Binary Relation Variables Viewed as Constrained Objects. In *Workshop on Logic Programming with Sets*, held alongside ICLP-1993.

[39] C. Gervet. Conjunto : Constraint Logic Programming with Finite Set Domains. In M. Bruynooghe, editor, *Proceedings of ILPS*-1994.

[40] C. Gervet. Interval Propagation to Reason about Sets: Definition and Implementation of a Practical Language. *In Constraints journal* 1(3), 1997.

[41] C. Gervet and P. Van Hentenryck. A New Set Domain Representation Using Length-Lex Ordering. Technical Report, TR-06-02, Brown University, 2006.

[42] K. Golden and W. Pang. Constraint Reasoning over Strings. In *Proceedings of CP*-2003.

[43] P. Hall. On Representatives of Subsets. *Journal of London Mathematical Society*, 10, 1935.

[44] P. Hawkins, V. Lagoon, and P.J. Stuckey. Set bounds and (split) set domain propagation using ROBDDs. In G. Webb and X. Yu, editors, *Proceedings of AI'04: Australian Joint Conference on Artificial Intelligence*, LNCS 3339, 2004.

[45] P. Hawkins, V. Lagoon, and P. Stuckey. Solving Set Constraint Satisfaction Problems using ROBDDs. *Journal of Artificial Intelligence Research* 24, 2005.

[46] N. Heintze and J. Jaffar. A Decision Procedure for a Class of SetConstraints. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.

[47] M. Hibti, H. Lombardi, and B. Legeard. Deciding in HFS-Theory via Linear Integer Programming with Application to Set Unification. In *Proceedings of LPAR*-1993.

[48] B. Hnich. *Function variables for Constraint Programming*. PhD thesis, Uppsala University,Department of Information Science, 2003.

[49] J. Hopcraft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Philippines, 1979.

[50] Ilog. User's manual. ILOG Solver 6.0 Sept., 2003.

[51] Ilog. User's manual. ILOG Configurator 2.3, 2004.

[52] S. Le Huitouze. A New Datastructure for Implementing Extensions to Prolog. In *Proceedings of PLILP*-1990, LNCS 456.

[53] D. Kapur and P. Narendran. NP-completeness of the set unification and matching problems. In *Proceedings of CADE*, 1986.

[54] Z. Kiziltan and T. Walsh. Constraint Programming with Multisets. In *Proceedings of the SymCon-02 workshop*, held alongside CP-2002.

[55] D. Knuth. The Art of Programming, Volume 4, Pre-Fascicle 2a: Generating all tuples.

[56] R.A. Kowalski. Predicate Logic as a Programming Language. In *Proceedings of IFIP*-1974.

[57] G. Kuper. *Logic Programming with Sets*, volume 41 of *1*, Academic Press, 1990.
[58] F. Laburthe. CHOCO: Implementing a CP Kernel. http://www.choco-constraints.net/, 2000. In *Proceedings of TRICS*, held alongside CP-2000.
[59] F. Laburthe. Constraints over Ontologies. In F. Rossi, editors, *Proceedings of CP*-2003.
[60] V. Lagoon and P.J. Stuckey. Set domain propagation using ROBDDs. In M. Wallace, editor, *Proceedings CP-2004*, LNCS 3258.
[61] J. L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10, 1978.
[62] J.H.M. Lee and H. van Emden. Interval Computation as Deduction in CHIP. In *Journal of Logic Programming*, 16 (3-4), Elsevier, 1993.
[63] B. Legeard and E. Legros. Short overview of the CLPS System. In *Proceedings of PLILP*-1991.
[64] C.C. Lindner and A. Rosa. *Topics on Steiner Systems*, volume 7 of *Annals of Discrete Mathematics*. North Holland, 1980.
[65] M. Livesey and J. Siekmann. Unification of Sets and Multisets. Memo seki-76-ii, University of St. Andrews (Scotland) and Universitât Karlsruhe (Germany) Department of Computer Science, 1976.
[66] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 1977.
[67] Mozart/Oz, `http://www.moxart-oz.org/`.
[68] T. Müller. *C*onstraint Propagation in Mozart. PhD dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, SaarbrÄucken, Germany, 2001.
[69] T. Müller and M. Müller. Finite Set Constraints in Oz. In *Workshop Logische Programmierung*, Burkhard Freitag and Dietmar Seipel, editors, 13, 1997.
[70] T. Müller. Solving Set Partitioning Problems with Constraint Programming. In *Proceedings of PAPPACT*-1998.
[71] W. Older and A. Vellino. Constraint Arithmetic on Real Intervals. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Papers*. MIT Press, 1993.
[72] L. Pacholski and A. Podelski. Set Constraints: a Pearl in Research and Constraints. Tutorial at CP-1997.
[73] K. J. Perry, K. V. Palem, K. MacAloon, and G. M. Kuper. The Complexity of Logic Programming with Sets. *Computer Science*, 1986.
[74] G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. In *Proceedings of CP*-2004.
[75] J-F. Puget. PECOS a High Level Constraint Programming Language In *Proceedings of Spicis*, 1992.
[76] J-F. Puget. Set Constraints and Cardinality Operator: Application to Symmetrical Combinatorial Problems. In *Third Workshop on Constraint Logic Programming (WCLP'93)*, 1993.
[77] J.F. Puget. Finite set intervals. In *Workshop on set constraints*, held alongside CP-1996.
[78] C.-G. Quimper and T. Walsh. Beyond Finite Domains: the All Different and Global Cardinality Constraints. in *Proc. of CP*-2005, 2005.
[79] J.C. Régin. Generalized arc consistency for global cardinality constraints. In *Proceedings of AAAI*-1996, AAAI Press/The MIT Press.

[80] J.C. Régin and J.-F. puget. A Filtering Algorithm for Global Sequencing Constraints. In *Proceedings of CP*-1997, LNCS.

[81] J.C. Reynolds. Automatic Computation of Data Set Definitions. *In Information Processing*, 68, 1969.

[82] A. Sadler and C. Gervet. Global Reasoning on Sets. In *FORMUL'01 workshop on modelling and problem formulation* held alongside CP-2001.

[83] A. Sadler and C. Gervet. Global Filtering for the Disjointness Constraint on Fixed Cardinality Sets. Technical report ICPARC-04-02, March 2004.

[84] A. Sadler and C. Gervet. Hybrid Set Domains to Strengthen Constraint Propagation and Reduce Symmetries. In M. Wallace, editor, *Proceedings of CP*-2004, LNCS.

[85] A. Sadler. Strengthening Finite Set Constraint Solvers through Active Use of Problem Structure, Symmetries and Cardinality Information. PhD thesis, University of London, Imperial College, April 2005.

[86] J. Schimpf, A. Cheadle, W. Harwey, A. Sadler, K. Shen, and M. Walllace. ECL^{i}PS^e Technical report 03-1, IC-Parc, Imperial College London, 2003.

[87] O. Shmueli, S. Tsur, and C. Zaniolo. Compilation of set terms in the logic data language (LDL). *The Journal of Logic Programming*, 12(12):89–119, 1992.

[88] B. M. Smith. Symmetry and Search in a Network Design Problem. In *Proceedings of CP-AI-OR*-2005, LNCS 3524, Springer, 2005.

[89] F. Stolzenburg. Membership-constraints and complexity in logic programming with sets. In Franz Baader and Klaus U. Schulz, editors, *Frontiers in Combining Systems*, Kluwer Academic, 1996.

[90] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. The MIT Press, 1989.

[91] P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

[92] C. Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *Proceedings of ICLP*-1989.

[93] T. Walsh. Consistency and Propagation with Multiset Constraints: A Formal Viewpoint. In *Proceedings of CP*-2003, LNCS.

[94] N.F. Zhou. B-Prolog http://www.probp.com/.