Constraints (2007) 12:93–129
DOI 10.1007/s10601-006-9012-6

Cardinal: A Finite Sets Constraint Solver

Francisco Azevedo

Published online: 17 February 2007

Abstract In this paper we present *Cardinal*, a general finite sets constraint solver just made publicly available in ECLiPSe Constraint System, suitable for combinatorial problem solving by exploiting inferences over sets cardinality. In fact, to deal with set variables and set constraints, existing set constraint solvers are not adequate to handle a number of problems, as they do not actively use important information about the cardinality of the sets, a key feature in such problems. *Cardinal* is formally presented as a set of rewriting rules on a constraint store and we illustrate its efficiency with experimental results. We show the importance of propagating constraints on sets cardinality, by comparing *Cardinal* with other solvers. Another contribution of this paper is on modelling: we focus essentially on digital circuits problems, for which we present new modelling approaches and prove that sets alone can be used to model these problems in a clean manner and solve them efficiently using *Cardinal*. Results on a set of diagnostic problems show that *Cardinal* obtains a speed up of about two orders of magnitude over *Conjunto*, a previous available set constraint solver, which uses a more limited amount of constraint propagation on cardinalities. Additionally, to further extend modelling capabilities and efficiency, we generalized *Cardinal* to actively consider constraints over set functions other than cardinality. The *Cardinal* version just released allows declaring union, minimum and maximum functions on set variables, and easily constraining those functions, letting *Cardinal* especial inferences efficiently take care of different problems. We describe such extensions and discuss its potentialities, which promise interesting research directions.

Keywords CLP, Constraint Logic Programming · PI, Primary Input · TG, Test Generation

1 Introduction

A set is naturally used to collect distinct elements sharing some property. Combinatorial search problems over these data structures can thus be naturally modelled by high level languages with set abstraction facilities, and efficiently solved if constraint reasoning prunes search space when the sets are not fully known a priori (i.e. they are variables ranging over a set domain).

F. Azevedo (✉)
Departamento de Informática, CENTRIA, FCT/UNL, Monte da Caparica, Portugal
e-mail: fa@di.fct.unl.pt

Set constraints have deserved in the last years special attention by the Constraint Programming community and have been addressed in recent literature for set-based program analysis systems for infinite sets [25] and, as finite set constraint relations, in languages for general set-based combinatorial search problems [6, 10]. Many interesting theoretical and practical results were obtained [11, 16, 20, 24, 27, 29] making it a very rich and promising research topic [1, 28].

Many complex relations between sets can be expressed with constraints such as set inclusion, disjointness and equality over set expressions that may include such operators as intersection, union or difference of sets. Also, as it is often the case, one is not interested simply on these relations but on some attribute or function of one or more sets (e.g. the cardinality of a set). For instance, the goal of many problems is to maximise or minimise the cardinality of a set. Even for satisfaction problems, some sets, although still variables, may be constrained to a fixed cardinality or a stricter cardinality domain than just the one inferred by the domain of a set variable (for instance, the cardinality of a set may have to be restricted to be an even number).

Finite set constraints were introduced in *PECOS* [29] and *Conjunto* [20] (formalized in [21]). These were the first languages to represent set variables by set intervals with a lower and an upper bound considering set inclusion as a partial ordering. Consistency techniques are then applied to set constraints by interval reasoning [8]. In *Conjunto* (available as an ECLiPSe [17] library), a set domain variable S is specified by an interval $[a, b]$ where a and b are known sets ordered by set inclusion, representing the greatest lower bound and the lowest upper bound of S, respectively.

To deal with optimisation problems, *Conjunto* includes the cardinality of a set as a graded function in the system, and generalises a graded function as $f: P(H_u) \rightarrow N$ mapping a non-quantifiable term of the power-set of the Herbrand universe to a unique integer value denoting a measure of the term, and satisfying $S_1 \subseteq S_2 \Rightarrow f(S_1) \leq f(S_2)$ for two sets s_1, s_2.

The cardinality of a set S, given as a finite domain variable C ($\#S=C$), is not a bijective function since two distinct sets may have the same cardinality. Still, due to the properties of a graded function, it can be constrained by the cardinalities of the set bounds. *Conjunto* allows graduated constraints over cardinalities but this cardinality information is largely disregarded until it is known to be equal to the cardinality of one of the set bounds, in which case an inference rule is triggered to instantiate the set.

Although *Conjunto* represented a great improvement over previous CLP languages with set data structures [20], it lacked some inferences on the cardinality level, which are crucial for a number of CSPs. In fact, *Conjunto* makes a very limited use of the information about the cardinality of set variables. The reason for this lies in the fact that it, is in general, too costly to derive all the inferences one might do over the cardinality information in order to tackle the problems *Conjunto* had initially been designed for (i.e. large scale set packing and partitioning problems) (Gervet, 1999, personal communication). Nonetheless, and given their nature, we anticipated that some use of this information could be quite useful and speed up the solving of these problems.

Recently, set solvers with domains using reduced ordered binary decision diagrams (ROBDDs) have been proposed [24, 27] with more efficient domain propagators, but that are not so efficient when handling cardinality constraints.

Inferences using cardinalities can be very useful to deduce more rapidly the non-satisfiability of a set of constraints, thus improving efficiency of combinatorial search problem solving. As a simple example, if Z is known to be the set difference between Y and X, both contained in set $\{a, b, c, d\}$, and it is known that X has exactly two elements, it should be inferred that the cardinality of Z can never exceed two elements (i.e. from

$X, Y \subseteq \{a, b, c, d\}$, $\#X=2$, $Z=Y\backslash X$ it should be inferred that $\#Z\leq 2$). A failure could thus be immediately detected upon the posting of a constraint such as $\#Z=3$.

We thus propose that set constraint solvers must handle the sets cardinality more actively, given the important role this feature plays in diagnostic related problems [2, 6]. Inference capabilities such as these are particularly important when solving set problems where cardinality plays a special role, as is the case of some circuit problems, as we discuss in this paper. We therefore developed a new constraint solver over sets [2, 6] that fully uses constraint propagation on sets cardinality, and which we generalise to other set functions [4]. Since these especial inferences were proposed, *Cardinal* has been improved, extended and made stable, and has just been made available as an ECLiPSe Prolog third party library, as of version 5.7 build #60. In this paper we comprehensively describe *Cardinal*'s operational semantics and current features. Additional possible extensions of *Cardinal* are also discussed for other applications such as timetabling and other scheduling problems.

To stress the advantages of cardinality reasoning we turn to some digital circuits problems, where we present a new modelling approach and conversion of extended digital signals to sets alone, which greatly simplifies previous models and drastically improves solving times when using *Cardinal*.

This paper is organised as follows: Section 2 describes *Cardinal* and the theory behind it; in Section 3 we show how we can model a number of digital circuits problems using set constraints; then, in Section 4, we show its usefulness with experimental results and comparisons with other approaches, also discussing other possible applications of *Cardinal*, in all its extension. We conclude in Section 5, discussing *Cardinal* potentialities and lines of future research.

2 Cardinal

Before describing *Cardinal* operational semantics and implementation, we will review some basic concepts of set theory that will be useful for its explanation.

2.1 Intervals and Lattices

Set intervals define a lattice [9, 22, 23] of sets. Figure 1 illustrates the powerset lattice for the example set domain $U=\{a, b, c, d\}$, where a line connecting set S_1 to underneath set S_2 means $S_2 \subseteq S_1$. The set inclusion relation $\subseteq$ between two sets defines a partial order on

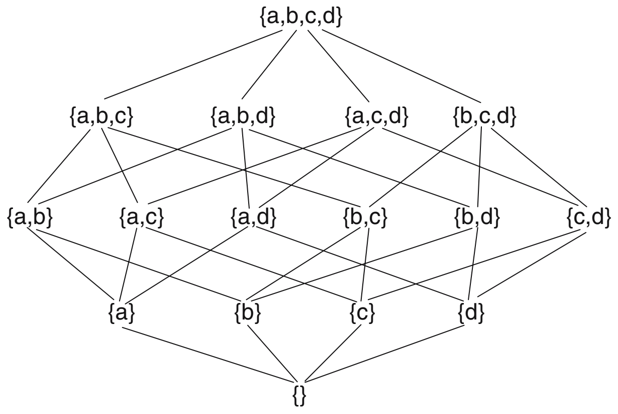


Fig. 1 Powerset lattice for $U=\{a, b, c, d\}$, with set inclusion as partial order

powerset $P(U)$, the set of all subsets of U. Hence $P(U)$ is a partially ordered set where binary relation $\subseteq$ has the following properties:

- $\forall S : S \subseteq S$ **Reflexivity**
- $\forall S, T : ((S \subseteq T) \wedge (T \subseteq S)) \Rightarrow (S = T)$ **Antisymmetry**
- $\forall S, T, U : ((S \subseteq T) \wedge (T \subseteq U)) \Rightarrow (S \subseteq U)$ **Transitivity**

There are thus other implicit inclusion relations in the lattice of Fig. 1 that were not explicitly drawn, due to the transitivity rule. The top set, U, includes all sets of $P(U)$; while the bottom set, $\{\}$, is included in all sets of $P(U)$. Consequently, sets U and $\{\}$ constitute an upper bound and a lower bound of $P(U)$, respectively. In addition, they are the *least upper bound* (*lub*) or join, and the *greatest lower bound* (*glb*) or meet of $P(U)$, since there is no other upper bound contained in ('less' than) U nor other lower bound containing ('greater' than) the empty set $\{\}$.

Let us now consider the sub-lattice of Fig. 2a. Sets $\{\}$ and $\{a, b, c, d\}$ are still a lower and an upper bound, but this time the *glb* is $\{b\}$ and the *lub* is $\{a, b, d\}$.

The two bounds (*glb* and *lub*) define a set interval (e.g. $[\{b\},\{a, b, d\}]$) and may form the domain of a set variable S, meaning that set S is one of those defined by its interval (lattice); all other sets outside this domain are excluded from the solution. Thus, b is definitely an element of S, while a and d are the only other possible elements (Venn diagram in Fig. 2b).

Set interval reasoning allows us to apply consistency techniques such as Bounds Consistency, due to the monotonic property of set inclusion.

Any set variable must then have a domain consisting of a set interval. In addition, this interval should be kept as small as possible, in order to discard all sets that are known not to belong to the solution, while not loosing any of the still possible values (sets). The smallest such domain is the one with equal *glb* and *lub*, i.e. a domain of the form $[B, B]$, corresponding to a constant set B. For a set variable that can assume any set value from a collection of known sets, such as $\{\{a, b\},\{a, c\},\{d\}\}$, the corresponding interval is the convex closure of such collection (which in this case is the set interval $[\{\},\{a, b, c, d\}]$). In general, for n possible arbitrary sets $S_1 ... S_n$, the corresponding set variable X has an interval domain $[glb, lub]$ where

$$glb = \bigcap_{i=1}^{n} S_i \quad \text{and} \quad lub = \bigcup_{i=1}^{n} S_i$$

2.2 Operational Semantics

In this section we describe *Cardinal*'s operational semantics showing constraint propagation over set variables with interval domains, together with inferences over sets' cardinality.

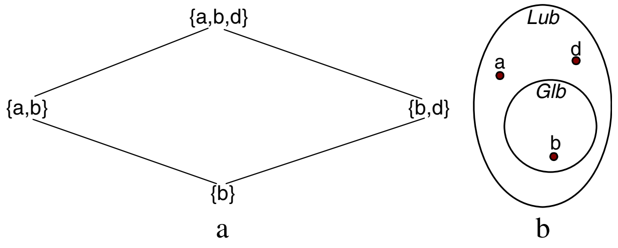


Fig. 2 Set interval $[\{b\},\{a, b, d\}]$: **a** Sub-lattice; **b** Venn diagram

The set universe notion is necessary not only for the set complement operation, but also for the especial cardinality inferences we propose. Hence we will use U to denote the set universe domain, and u to denote the cardinality of U (u=# U).

A set variable X is represented by $[a_X, b_X]_{Cx:Dx}$ (or simply as $[a_X, b_X]_{Cx}$) where a_X is its greatest lower bound (i.e. the elements known to belong to X), b_X its lowest upper bound (i.e. the elements not excluded from X), and C_X its cardinality (a finite domain variable) with domain D_X. In the remainder, a_X, b_X, C_X and D_X will be used to refer to these attributes of set variable X if no confusion arises.

Set variables whose only two possible values are the empty set $\varnothing$ (C_X=0) and the universe U (C_X=u), shown in gray in Fig. 3a, are then represented by $[\varnothing, U]_{Cx:\{0,\ u\}}$ (such situation may occur often in practice, as we describe in examples below). We may view powerset lattices as a number of horizontal layers of different cardinality; the set cardinality variable constrains possible layers where the solution value is. Figure 3b shows the lattice for set variable X with domain $[\{\},\{a, b, c, d\}]_1$ corresponding to a singleton. Such domain is sufficient to express the disjunction $X=\{a\} \vee X=\{b\} \vee X=\{c\} \vee X=\{d\}$ and, therefore, one does not need to explicitly post this disjunctive constraint.

Cardinal implements a number of set constraints such as inclusion, equality, inequality, membership and disjointness, together with set operations (union, intersection, difference and complement), as built-in. (Nonetheless, the equality and inclusion constraints together with the operations of sets complement and binary intersection are sufficient to model set problems.) We will next describe all these *Cardinal* constraints. Inferences will be formally described as rewriting rules as in the following schematic figure:

$$(\text{trigger condition})\frac{\text{pre} - \text{conditions}}{\text{CS\_changes}}$$

where CS refers to the Constraint Store. In addition, a number on the right identifies each inference rule.

The constraint store maintains constraints over sets and over finite domains (the cardinality of the sets), but we only describe the rewriting rules of the set constraints (we assume that a finite domain constraint solver maintains bounded arc-consistency, or interval consistency, on these constraints).

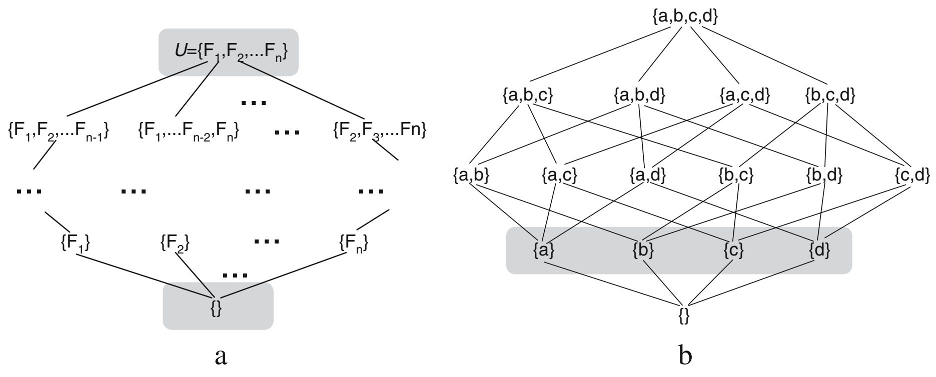


Fig. 3 Powerset lattices with cardinalities: **a** circuit PI; **b** singleton

2.2.1 Set Variable

When a variable is declared as a set variable, it is simply included in the constraint store, ensuring the bounds of the variable and that its cardinality C is a finite domain variable with domain D:

$$\frac{}{\{\text{tell}\big(X \in [a,b]_{C:D}\big)\}\mapsto\{X \in [a,b]c, C :: D\}} \tag{1}$$

A number of inferences are subsequently maintained. During computation, set intervals can only get shorter, i.e. either because the lower bound becomes larger or the upper bound smaller, or both.

The cardinality must always remain inside the limits given by the set bounds (the triggers of these inferences are shown in parenthesis next to the rewriting rules, and may correspond to one or more variables becoming ground, changing bounds, or being bound in the Prolog sense):

$$(X : \text{changed bounds}) \qquad \frac{n = \#a_x, m = \#b_x}{\{\}\mapsto\{Cx \geq n, Cx \leq m\}} \tag{2}$$

As in *Conjunto*, a set variable becomes one of its bounds if their cardinality is the same (this rule is triggered when C_X becomes a fixed value and, afterwards, when X is bounded):

$$(C_X\text{: ground, or } X : \text{changed bounds}) \quad \frac{C_x = \#a_x}{\{\}\mapsto\{X = a_x\}} \; \frac{C_x = \#b_x}{\{\}\mapsto\{X = b_x\}} \tag{3}$$

When there are two domains declared for the same set variable, their intersection must be computed and cardinalities made equal:

$$\frac{a = a_1 \cup a_2, b = b_1 \cap b_2}{\{X \in [a_1,b_1]c_1, X \in [a_2,b_2]c_2\}\mapsto\{X \in [a,b]c_1, C_1 = C_2\}} \tag{4}$$

Eventually a failure may be detected, either because the lower bound of a set is not included in its upper bound, or the domain of the cardinality becomes empty:

$$(X : \text{changed bounds}) \qquad \frac{\text{not}(a_x \subseteq b_x)}{\{\}\mapsto\text{fail}} \quad \frac{D_x = \emptyset}{\{\}\mapsto\text{fail}} \tag{5}$$

2.2.2 Membership Constraints

Membership constraint is trivially handled by inserting, as soon as it is ground, the given element in the set's *glb*:

$$\frac{}{\{\text{tell}(\text{elem} \in X)\}\mapsto\{\text{elem} \in X\}} \tag{6}$$

$$(\text{elem} : \text{ground}) \qquad \frac{a = a_x \cup \{\text{elem}\}}{\{\text{elem} \in X\}\mapsto\{X \in [a, b_x]\}} \tag{7}$$

Its negation is similarly handled by removing the element from the *lub*:

$$\frac{}{\{\text{tell}(\text{elem} \notin X)\}\mapsto\{\text{elem} \notin X\}} \tag{8}$$

$$(\text{elem} : \text{ground}) \qquad \frac{b = b_x \backslash \{\text{elem}\}}{\{\text{elem} \notin X\}\mapsto\{X \in [a_x, b]\}} \tag{9}$$

2.2.3 Set Complement

Cardinality, u, of the universe (given explicitly or implicitly by the union of the sets' upper bounds) is used in a finite domain constraint, $C_Y = u - C_X$, over the sets cardinalities when the set complement constraint is posted. In general, the finite domains constraint solver maintains bounds consistency on this constraint. Nevertheless, we ensure full arc consistency when the constraint is posted:

$$\frac{}{\{\mathrm{tell}(X = \overline{Y})\} \mapsto \{Cy = u - C_x, X = \overline{Y}\}} \tag{10}$$

Only in an empty universe can a set be the same as its complement:

$$(X \,\mathrm{or}\, \mathrm{Y} : \mathrm{bound}) \quad \frac{}{\{X = \overline{Y}, X = Y\} \mapsto U = \emptyset} \tag{11}$$

Whenever there is an update of the bounds of one of the sets, the bounds of its complement must also be updated accordingly, which eventually leads to the successful instantiation of the sets or to a failure:

$$(X : \mathrm{changed\ bounds}) \quad \frac{a = \overline{b_x}, b = \overline{a_x}}{\{X = \overline{Y}\} \mapsto \{X = \overline{Y}, Y \in [a, b]\}} \tag{12}$$

$$(Y : \mathrm{changed\ bounds}) \quad \frac{a = \overline{b_y}, b = \overline{a_y}}{\{X = \overline{Y}\} \mapsto \{X = \overline{Y}, X \in [a, b]\}} \tag{13}$$

The constraint disappears (is trivially proved or disproved) when sets are ground and their complementary nature can be easily checked:

$$(X \text{ or } Y : \mathrm{ground}) \quad \frac{\mathrm{ground}(X), \mathrm{ground}(Y), a_x = \overline{a_y}}{\{X = \overline{Y}\} \mapsto \{\}} \quad \frac{\mathrm{ground}(X), \mathrm{ground}(Y), a_x \neq \overline{a_y}}{\{X = \overline{Y}\} \mapsto \mathrm{fail}} \tag{14}$$

2.2.4 Set Equality

When two sets are told to be equal, so does their cardinality:

$$\frac{}{\{\mathrm{tell}(X = Y)\} \mapsto \{X = Y, C_x = C_y\}} \tag{15}$$

When one bound of the set is updated, so does the corresponding bound of any set equal to it (the situation is similar to that of having two domains for the same set, as in rule (4)):

$$(X \,\mathrm{or}\, \mathrm{Y} : \mathrm{changed\ bounds})$$

$$\frac{a = a_x \cup a_y, b = b_x \cap b_y}{\{X = Y, X \in [a_x, b_x], Y \in [a_y, b_y]\} \mapsto \{X = Y, X \in [a, b], Y \in [a, b]\}} \tag{16}$$

Again, when sets are ground, the equality is easily confirmed (or infirmed):

$$(X \text{ or } Y : \mathrm{ground}) \quad \frac{\mathrm{ground}(X), \mathrm{ground}(Y), X = Y}{\{X = Y\} \mapsto \{\}} \quad \frac{\mathrm{ground}(X), \mathrm{ground}(Y), X \neq Y}{\{X = Y\} \mapsto \mathrm{fail}} \tag{17}$$

Of course, if only one of the sets becomes ground, the previous rule (16) enforces the other set either to become with the same bounds (and ground, in which case this rule eliminates the equality constraint) or with an empty domain, causing a failure.

2.2.5 Set Inequality

When two sets are told to be different, we cannot relate their cardinalities since these can be equal, even with different sets:

$$\frac{}{\{\text{tell}(X \neq Y)\} \mapsto \{X \neq Y\}} \tag{18}$$

Of course, the two sets cannot be the same:

$$(X \text{ or Y : bound}) \quad \frac{}{\{X \neq Y, X = Y\} \mapsto \text{fail}} \quad \frac{\text{ground}(X), \text{ground}(Y), X \neq Y}{\{X \neq Y\} \mapsto \{\}} \tag{19}$$

The inequality constraint does not allow much propagation; hence rule (19) is triggered only when a set is bound. Then, to check whether the constraint is satisfied, both sets must be ground. Notice that is not worth performing extra computation efforts just to check entailment, since no pruning will occur: a constraint will just be removed from the store.

2.2.6 Disjointness

When two sets are told to be disjoint ($X \$ Y$, meaning $X \cap Y = \varnothing$), the sum of their cardinalities cannot exceed that of the union of their upper bounds (the implicit universe):

$$\frac{u = \#\left(b_x \cup b_y\right)}{\{\text{tell}(X \$ Y)\} \mapsto \left\{X \$ Y, C_x + C_y \leq u\right\}} \tag{20}$$

Elements definitely in one set cannot belong to the other:

$$(X : \text{changed glb}) \quad \frac{b = b_y \backslash a_x}{\{X \$ Y\} \mapsto \left\{X \$ Y, Y \in \left[a_y, b\right]\right\}} \tag{21}$$

$$(Y : \text{changed glb}) \frac{b = b_x \backslash a_y}{\{X \$ Y\} \mapsto \{X \$ Y, X \in [a_x, b]\}} \tag{22}$$

If the two sets are the same, they must have no elements (empty set):

$$(X \text{ or } Y : \text{bound}) \quad \frac{}{\{X \$ Y, X = Y\} \mapsto \{X = \varnothing\}} \tag{23}$$

With the previous rules assured, one set becoming ground is a sufficient condition to remove the constraint, since all its elements will have been removed from the other set:

$$(X \text{ or } Y : \text{bound}) \quad \frac{\text{ground}(X) \vee \text{ground}(Y)}{\{X \$ Y\} \mapsto \{\}} \tag{24}$$

2.2.7 Set Inclusion

If Y contains X, then C_Y is greater (or equal) than C_X:

$$\frac{}{\{\text{tell}(X \subseteq Y)\} \mapsto \left\{X \subseteq Y, C_x \leq C_y\right\}} \tag{25}$$

When the lower bound (*glb*) of X increases, the lower bound of Y may also increase; and when the upper bound (*lub*) of Y decreases, so might happen to X:

$$(X : \text{changed glb}) \quad \frac{a = a_x \cup a_y}{\{X \subseteq Y\} \mapsto \{X \subseteq Y, Y \in [a, b_y]\}} \tag{26}$$

$$(Y : \text{changed lub}) \quad \frac{b = b_x \cap b_y}{\{X \subseteq Y\} \mapsto \{X \subseteq Y, X \in [a_x, b]\}} \tag{27}$$

If b_X is contained in a_Y, or X is equal to Y, the constraint $X \subseteq Y$ is trivially satisfied, and can be eliminated from the store:

$$(X \text{ or } Y : \text{bound}) \quad \frac{\text{ground}(X) \vee \text{ground}(Y), b_x \subseteq a_y}{\{X \subseteq Y\} \mapsto \{\}} \quad \frac{}{\{X \subseteq Y, X = Y\} \mapsto \{X = Y\}} \tag{28}$$

2.2.8 Set Intersection

While for the set complement the universe is usually explicitly given, for the intersection Z of sets X and Y, the universe can be considered as the union of the upper bounds ($U = b_X \cup b_Y$), with cardinality u (as already used for set disjointness in rule (20)).

Whenever there are two set variables involved in a constraint, their interval domains, if depicted in a Venn diagram (Fig. 4), define eight disjoint distinguishing sets of interest, each possibly empty. For instance, zone 6 of Fig. 4 corresponds to the elements that are definitely part of both sets X and Y; in zone 4 are the elements that definitely belong to X but can never belong to Y; in zone 2 we have the elements that can belong to X or Y but are not yet definite elements of either. Constraint propagation over these constraints must take into account all set zones in addition to the two cardinality domains.

The following rule states that the intersection of two sets must be contained in both sets, and posts a special constraint on the cardinality of the set intersection:

$$\frac{}{\{\text{tell}(Z = X \cap Y)\} \mapsto \{Z = X \cap Y, \text{tell}(Z \subseteq X), \text{tell}(Z \subseteq Y), \text{tell}(C_z = X \otimes Y)\}} \tag{29}$$

The special cardinality constraint over sets (C_Z=$X \otimes Y$) ensures that each possible value for C_Z has a supporting cardinality pair in domains D_X and D_Y when the intersection is posted. Before formalising this operation, we first analyse what can the domain of cardinality C_Z be. If we take possible cardinality values cx of D_X and cy of D_Y, and the sum of cx and cy exceeds u, there must be common elements to X and Y, and their intersection has at least $cx + cy - u$ elements. This 'worst' case is illustrated in Fig. 5, where we 'push' all elements of X to the left and elements of Y to the right.

To reason about the upper bound, C_Z can never exceed cx nor cy since Z is the sets intersection. The elements in a_X not in b_Y (i.e. $a_X \setminus b_Y$, corresponding to zone 4 of Fig. 4) can safely be subtracted from cx since they are definitely not part of the intersection, but are counted in cx (so an upper bound can be $(cx - \#(a_X \setminus \mathrm{b_Y}))$). A similar reasoning may be done for Y, yielding another upper bound. A final upper bound can thus be considered the minimum of the two (i.e. $\min(cx - \#(a_X \setminus \mathrm{b_Y}), cy - \#(a_Y \setminus b_X))$).

Thus, for each pair cx and cy, an integer range for C_Z is calculated, and the ranges for all such pairs are eventually merged. This can in fact be regarded as maintaining arc-consistency on the cardinality of X, Y and Z, when $Z = X \cap Y$. In fact, this arc-consistency is only enforced when the constraint is first told:

$$\frac{}{\{\text{tell}(C_z = X \otimes Y)\} \mapsto \{C_z \in \{n : \exists i \in D_x, j \in D_y, i + j - u \leq n \leq \min(i - \#(a_x \setminus b_y), j - \#(a_y \setminus b_x))\}\}} \tag{30}$$

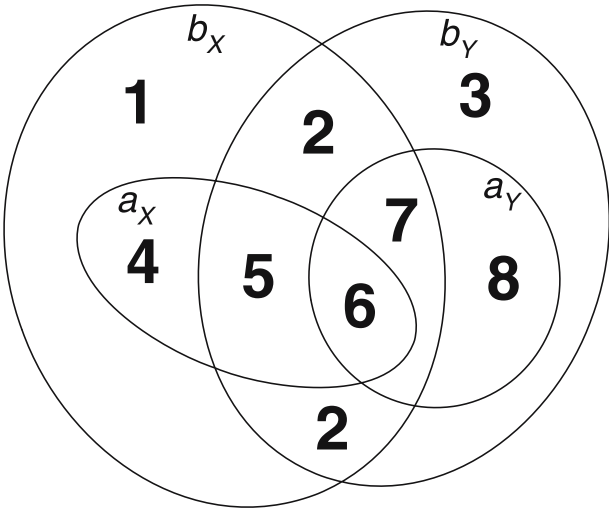


Fig. 4 Two sets, *X*, *Y*, define eight different zones

The usefulness of this rule can be illustrated with the following situation. Given two sets *X* and *Y* which can both be ∅ or *D*={f, g}, let us consider their intersection (this is a typical case with some digital circuits problems when two input bits are connected through an and-gate, as we will see in the examples section). While their set domain is the convex closure of the two bounds, their cardinality can only be 0 or 2. To find the cardinality domain of their intersection we examine cardinality pairs <*cx*, *cy*> = <0, 0>, <0, 2>, <2, 0>, and <2, 2>. The three first pairs yield only value 0 as a possible intersection cardinality, since one set has no elements and acts as an upper bound. Pair <2, 2> yields single value 2, since *u* is also 2 $(cx + cy - u = 2 + 2 - 2 = 2)$ as lower bound. Thus the final cardinality domain for the sets intersection is also {0, 2}. If only interval reasoning were performed on cardinality, the result would be the full range {0, 1, 2}.

Rather than checking pairs of integers, it is equivalent and more efficient to check pairs of sub-ranges and their bounds when the constraint is posted. Nevertheless, since this arc consistency is very costly to maintain, it is only checked when the constraint is posted. Subsequently, only bounds consistency is maintained on the cardinality of the sets by the underlying finite domains constraint solver:

$$(X : \text{changed glb or } Y\text{: changed lub}) \quad \frac{n = \#\left(a_x \backslash b_y\right)}{\{Z = X \cap Y\} \mapsto \{Z = X \cap Y, C_z \leq C_x - n\}} \quad (31)$$

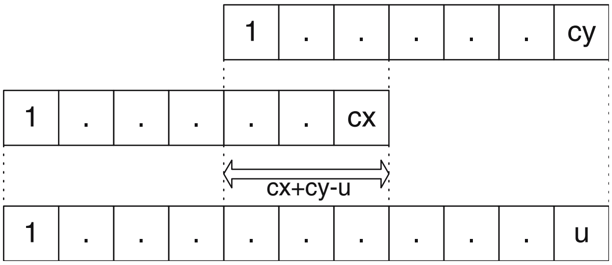


Fig. 5 Minimum intersection cardinality

$$(X:\text{changed lub or }Y\text{: changed lub})\quad \frac{n=\#(a_y\backslash b_x)}{\{Z=X\cap Y\}\mapsto\{Z=X\cap Y, C_z\le C_y-n\}} \tag{32}$$

$$(X\text{ or }Y:\text{changed lub})\quad \frac{u=\#(b_x\cup b_y)}{\{Z=X\cap Y\}\mapsto\{Z=X\cap Y, C_z\ge C_x+C_y-u\}} \tag{33}$$

A number of other inferences are performed regarding intersection. The lower bound of the set intersection is kept as the intersection of the lower bounds of the arguments (zone 6 of Fig. 4):

$$(X\text{ or }Y:\text{changed glb})\quad \frac{a=a_x\cap a_y}{\{Z=X\cap Y\}\mapsto\{Z=X\cap Y, Z\in[a,b_z]\}} \tag{34}$$

If both arguments are the same set, their intersection is that set (idempotence):

$$(X\text{ or }Y:\text{bound})\quad \frac{}{\{Z=X\cap Y, X=Y\}\mapsto\{X=Y,\text{tell}(Z=X)\}} \tag{35}$$

If intersection Z is known to be the same set as one of its arguments, then the intersection constraint may be eliminated (as $Z\subseteq X$ and $Z\subseteq Y$):

$$(X\text{ or Y}:\text{bound})$$
$$\frac{}{\{Z=X\cap Y, X=Z\}\mapsto\{X=Z\}}\quad\frac{}{\{Z=X\cap Y, Y=Z\}\mapsto\{Y=Z\}} \tag{36}$$

Conversely, if an argument contains the other, the intersection is the included set:

$$(X\text{ or }Y:\text{bound})\quad \frac{\text{ground}(Y), b_x\subseteq a_y}{\{Z=X\cap Y\}\mapsto\{\text{tell}(Z=X)\}}\quad\frac{\text{ground}(X), b_y\subseteq a_x}{\{Z=X\cap Y\}\mapsto\{\text{tell}(Z=Y)\}} \tag{37}$$

Although inclusion could be inferred more generally, for efficiency reasons this rule is only checked when either one of the arguments is ground. These four simplification/ simpagation rules [18] exploit the fact that the universe is the neutral element of the intersection. Here, the universe is the set argument containing the other.

All common elements to X and Y must be in Z. That is, X and Y must have no common elements outside Z. Hence, definite elements of argument X (Y) that are impossible in intersection Z must be removed from the other argument Y (X). This is a costly operation, so it is performed only on instantiation of variables:

$$(X\text{ or }Z:\text{ground})\quad \frac{b=b_y\backslash(a_x\backslash b_z)}{\{Z=X\cap Y\}\mapsto\{Z=X\cap Y, Y\in[a_y,b]\}} \tag{38}$$

$$(Y\text{ or }Z:\text{ground})\quad \frac{b=b_x\backslash(a_y\backslash b_z)}{\{Z=X\cap Y\}\mapsto\{Z=X\cap Y, X\in[a_x,b]\}} \tag{39}$$

In addition to the especial inferences exploiting the idempotence property and neutral element of set intersection (rules (35), (36) and (37)), which are triggered when some set is bound, let us now try to schematise inferences on variable bounds.

Globally, as to set bounds are concerned, we can make the inferences of Table 1, where some change in the set bound in the first column may have as effect an update of another set variable bound. Entries of the table thus express the update of the bound corresponding

Table 1 Set intersection: cause–effect rules on set bounds

Effect **Change**	a_x	b_x	a_y	b_y	a_z	b_z	
a_x			–	$-(a_x \backslash b_z)$	$a_x \cap a_y$	–	38, 34
b_x			–	–	–	b_x	$Z \subseteq X$
a_y	–	$-(a_y \backslash b_z)$			$a_x \cap a_y$	–	39, 34
b_y	–	–			–	b_y	$Z \subseteq Y$
a_z	a_z	–	a_z	–			$Z \subseteq X, Z \subseteq Y$
b_z	–	$-(a_y \backslash b_z)$	–	$-(a_x \backslash b_z)$			39, 38
	$Z \subseteq X$	39	$Z \subseteq Y$	38	34	$Z \subseteq X, Z \subseteq Y$	**Rules**

to its column due to some change of the bound corresponding to its line. Since lower bounds can only become larger, their updates are given in the form of a set of mandatory elements (i.e. another lower bound)—the final lower bound will thus be the union of the original one with this new one. Conversely, upper bounds become shorter and entry values correspond to another upper bound, which forces the intersection of the two upper bounds to obtain the final *lub*. Alternatively, upper bound updates may be given in the form-*(Set)* meaning that the elements of *Set* must be removed from the final *lub*.

Each line also includes the inference rules that cover the updates due to its corresponding bound change. Similarly, each column includes the rules that cover the updates of its corresponding bound. To find the rule that covers a particular update given by some table entry, one just has to intersect the rules of its line with the rules of its column.

For example, from a change in a_x (i.e. inclusion of elements in X) one may infer that elements $(a_x \backslash b_z)$ must be removed from b_y (i.e. cannot be part of Y). The same happens when there is a change in b_z (i.e. removal of elements from Z). These relations are covered by rule (38), which however is only triggered when X or Z is ground, for efficiency reasons.

As we can see from the table, each bound change affects some other bound, and each bound may be affected due to a change in another set variable bound. Among the rules that cover these updates are also the inclusion constraints (described above) that are posted when the set intersection constraint is posted (rule (29)).

Rules (38) and (39) wait for some instantiation, while rule (34) and the inclusion constraint are immediately triggered by the set bound change.

Of course, bound changes also affect cardinalities (e.g. rule (2)), and cardinalities affect bounds (rule (3)) in addition to being mutually dependent. These relations are handled by the constraints:

$$\begin{aligned} C_z &\leq C_x - \#\left(a_x \backslash b_y\right) \\ C_z &\leq C_y - \#\left(a_y \backslash b_x\right) \\ C_z &\geq C_x + C_y - \#\left(b_x \cup b_y\right) \end{aligned}$$

corresponding to rules (31), (32) and (33) (also, cardinalities are nonnegative and limited by set bounds). After reaching arc consistency upon the posting of some set intersection, the constraint solver applies Bounds Consistency on these cardinality constraints, which ensures that the bounds of cardinality domains are updated. In addition, such constraints are also updated due to changes in set bounds, which may further tighten the domains (e.g. when a_x increases or b_y decreases, $\#(a_x \backslash b_y)$ may increase and the upper bound of Cz given by $C_x - \#\left(a_x \backslash b_y\right)$ decreases, which further constrains Cz, and possibly Cx whose lower bound is then $C_z + \#\left(a_x \backslash b_y\right)$).

2.2.9 Set Union

For the union *Z* of sets *X* and *Y*, we can find many similarities with the intersection: the universe can also be considered as the union of the upper bounds ($U = b_X \cup b_Y$), and *u* its cardinality.

The union of two sets, *X* and *Y*, must contain both sets, and its cardinality is not simply the sum of C_X and C_Y, since *X* and *Y* may have common elements. We know then that it can never exceed C_X+C_Y. In addition, the union constraint posts another special constraint on the resulting cardinality:

$$\frac{}{\{\text{tell}(Z = X \cup Y)\} \mapsto \{Z = X \cup Y, \text{tell}(X \subseteq Z), \text{tell}(Y \subseteq Z), C_z \leq C_x + C_y, \text{tell}(C_z = X \oplus Y)\}} \tag{40}$$

Before formalising the cardinality constraint ($C_Z=X\oplus Y$), let us again first analyse what the domain of cardinality C_Z can be. Given possible cardinalities *cx* of D_X and *cy* of D_Y, C_Z can never exceed *cx*+*cy* nor *u*, and the upper bound is thus min(*cx*+*cy*, *u*). To find a lower bound, since *Z* is the sets union, it contains *X* and has at least the *cx* elements. To these we can safely add the elements in a_Y not in b_X (i.e. $a_Y \backslash b_X$, corresponding to zone 8 of Fig. 4), since they are definitely part of the union and are not counted in *cx*. Similar reasoning is done for *Y*, yielding another lower bound, so the maximum of the two is the final lower bound given the <*cx*, *cy*> pair (i.e. $\max(cx + \#(a_Y \backslash b_X), cy + \#(a_X \backslash b_Y))$). Thus, for each such pair, an integer range for C_Z is calculated when the constraint is first told:

$$\frac{}{\{\text{tell}(C_z = X \oplus Y)\} \mapsto \{C_z \in \{n : \exists i \in D_x, j \in D_y, \max\left(i + \#(a_y \backslash b_x), j + \#(a_x \backslash b_y)\right) \leq n \leq \min(i+j, u)\}\}} \tag{41}$$

As an example, let us take two sets *X* and *Y* that can only be ∅ or {*f*, *g*, *h*, *i*} (cardinality 0 or 4). To find the cardinality domain of their union we examine cardinality pairs <0, 0>, <0, 4>,<4, 0> and <4, 4>. The three last pairs yield only value 4 as a possible union cardinality, since at least one set has four elements, which is also the maximum possible value (*u*). Obviously, pair <0, 0> yields single value 0. Thus the final cardinality domain for the sets union is also {0, 4}. This is a typical case for the circuit models below with two inputs passing through an or-gate. As with set intersection, if only interval reasoning were performed on the cardinality, the result would be the full range 0...4 and no propagation would be achieved for the rest of the circuit.

Subsequently, bounds consistency is maintained on the cardinality of the sets:

$$(X : \text{changed glb or Y : changed lub}) \quad \frac{n = \#(\mathrm{a_x} \backslash \mathrm{b_y})}{\{Z = X \cup Y\} \mapsto \{Z = X \cup Y, C_z \geq C_y + n\}} \tag{42}$$

$$(X : \text{changed lub or Y : changed glb}) \quad \frac{n = \#(a_y \backslash b_x)}{\{Z = X \cup Y\} \mapsto \{Z = X \cup Y, C_z \geq C_x + n\}} \tag{43}$$

Z's cardinality upper bound is updated by rule (2), whenever its *lub* changes.

As to set bounds, the upper bound of the set union is kept as the union of the upper bounds of the arguments:

$$(X \text{ or } Y : \text{changed lub}) \quad \frac{b = b_x \cup b_y}{\{Z = X \cup Y\} \mapsto \{Z = X \cup Y, Z \in [a_z, b]\}} \tag{44}$$

If both arguments are the same set, their union is that set (idempotence):

$$(X \text{ or } Y : \text{bound}) \quad \frac{}{\{Z = X \cup Y, X = Y\} \mapsto \{X = Y, \text{tell}(Z = X)\}} \tag{45}$$

If union Z is known to be the same set as one of its arguments, then the union constraint may be eliminated (since $X \subseteq Z$ and $Y \subseteq Z$):

$$(X \text{ or } Y : \text{bound}) \quad \frac{}{\{Z = X \cup Y, X = Z\} \mapsto \{X = Z\}} \quad \frac{}{\{Z = X \cup Y, Y = Z\} \mapsto \{Y = Z\}} \tag{46}$$

Conversely, if an argument contains the other, the union is the container set:

$$(X \text{ or } Y : \text{bound}) \quad \frac{\text{ground}(Y), b_y \subseteq a_x}{\{Z = X \cup Y\} \mapsto \{\text{tell}(Z = X)\}} \quad \frac{\text{ground}(X), b_x \subseteq a_y}{\{Z = X \cup Y\} \mapsto \{\text{tell}(Z = Y)\}} \tag{47}$$

These four rules exploit the fact that the universe (the set argument containing the other) is the absorbing element of the sets' union.

All elements of X and Y must be in Z. Thus, if Z has elements outside X (Y), then those elements must belong to Y (X). As a costly operation, it is performed only on instantiation of variables:

$$(X \text{ or } Z : \text{ground}) \quad \frac{a = a_y \cup (a_z \backslash b_x)}{\{Z = X \cup Y\} \mapsto \{Z = X \cup Y, Y \in [a, b_y]\}} \tag{48}$$

$$(Y \text{ or } Z : \text{ground}) \quad \frac{a = a_x \cup (a_z \backslash b_y)}{\{Z = X \cup Y\} \mapsto \{Z = X \cup Y, X \in [a, b_x]\}} \tag{49}$$

Inferences on set bounds and corresponding rules for the set union constraint are depicted in Table 2, similarly to Table 1 for set intersection.

Inclusion constraints are posted by rule (40) (when union constraint is told).Cardinality constraints are:

$$\begin{aligned} C_z &\leq C_x + C_y \\ C_z &\geq C_y + \#(a_X \backslash b_Y) \\ C_z &\geq C_x + \#(a_Y \backslash b_X) \end{aligned}$$

corresponding to rules (40), (42), and (43).

2.2.10 Set Difference

For the difference Z of sets X and Y ($Z = X \backslash Y$), the universe is, as always, the union of the upper bounds ($U = b_X \cup b_Y$), and u its cardinality.

The result Z of removing Y from X, must be contained in X and disjoint from Y, and its cardinality is not simply the difference of C_X and C_Y, since elements of Y may be absent from X. We know then that it is at least $C_X - C_Y$. In addition, the difference constraint posts another special constraint on the resulting cardinality ($C_Z = X \div Y$):

$$\frac{}{\{\text{tell}(Z = X \backslash Y)\} \mapsto \{Z = X \backslash Y, \text{tell}(Z \subseteq X), \text{tell}(Y \$ Z), C_z \geq C_x - C_y, \text{tell}(C_z = X \div Y)\}} \tag{50}$$

Table 2 Set union: cause–effect rules on set bounds

Effect Change	a_x	b_x	a_y	b_y	a_z	b_z	
a_x			–	–	a_x	–	$X \subseteq Z$
b_x			$a_z \backslash b_x$	–	–	$b_x \cup b_y$	48, 44
a_y	–	–			a_y	–	$Y \subseteq Z$
b_y	$a_z \backslash b_y$	–			–	$b_x \cup b_y$	49, 44
a_z	$a_z \backslash b_y$	–	$a_z \backslash b_x$	–			49, 48
b_z	–	b_z	–	b_z			$X \subseteq Z, Y \subseteq Z$
	49	$X \subseteq Z$	48	$Y \subseteq Z$	$X \subseteq Z, Y \subseteq Z$	44	**Rules**

Let us then analyse what values can C_Z assume if we take possible cardinality values cx of D_X and cy of D_Y. It is at least $cx-cy$, as explained, since at most cy elements will be removed from X. Also, we can subtract from cx at most the number of common elements to both *lub*s, i.e. $\#(a_X \cap a_Y)$. Hence, Z's cardinality is at least $cx - \#(b_X \cap b_Y)$. Joining the two, we have the lower bound $\max(cx - cy, cx - \#(b_X \cap b_Y))$.

As to the upper bound, since at least cy elements will not be a part of Z, we know that it contains at most $u-cy$ elements. It can also never exceed cx. Furthermore, from cx we can safely subtract the definite common elements from X and Y, i.e. $\#(a_X \cap a_Y)$. The upper bound is thus $\min(cx - \#(a_X \cap a_Y), u - cy)$ and we can formalise the posting of the cardinality constraint as follows:

$$\frac{}{\{\text{tell}(C_z = X \div Y\} \mapsto \{C_z \in \{n : \exists i \in D_x, j \in D_y, \max(i-j, i-\#(b_x \cap b_y)) \leq n \leq \min(i - \#(a_x \cap a_y), u-j)\}\}} \tag{51}$$

We take again the two example sets X and Y that can only be $\varnothing$ or $\{f,g,h,i\}$ (cardinality 0 or 4). To find the cardinality domain of their difference $Z = X \backslash Y$, we examine cardinality pairs <0, 0>, <0, 4>,<4, 0> and <4, 4>. When cy=4, the resulting cardinality must be 0 (upper bound given by $u-cy$) since u is also 4. If cx=0, the resulting cardinality must also be 0 (bounded by $cx - \#(a_X \cap a_Y)$). The remaining pair, <4, 0>, yields only value 4 as a possible difference cardinality, since no elements are removed from set X with four elements ($\#Z$ is lower bounded by $cx-cy$). The final cardinality domain for the sets difference is thus also $\{0, 4\}$.

For another example with different sets and cardinalities, let us take $X \in [\{\},\{e1, e2, \ldots e20\}]_{10}$, and $Y \in [\{\},\{e20, e21, \ldots e30\}]$. In this case we may conclude that the cardinality of $Z = X \backslash Y$ has at least nine elements due to the lower bound $cx - \#(b_X \cap b_Y) = 10 - \#\{e20\} = 10 - 1 = 9$. The domain of the cardinality of Z is then constrained to $\{9, 10\}$ using just this rule.

Let us now consider two sets X, Y with domain $[\{e1, e2, \ldots e5\},\{e1, e2, \ldots e20\}]_6$. Their difference has at most one element due to the bound $cx = -\#(a_X \cap a_Y)$. This rule thus allows us to constrain the cardinality of difference Z to the simple domain $\{0, 1\}$ from start.

Subsequently, as with the intersection and union operations, bounds consistency is applied on the cardinality of the sets:

$$(X \text{ or } Y : \text{changed glb}) \quad \frac{n = \#(a_x \cap a_y)}{\{Z = X \backslash Y\} \mapsto \{Z = X \backslash Y, C_z \leq C_x - n\}} \tag{52}$$

$$(X \text{ or } Y : \text{changed lub}) \quad \frac{n = \#(b_x \cup b_y)}{\{Z = X \backslash Y\} \mapsto \{Z = X \backslash Y, C_z \leq n - C_y\}} \tag{53}$$

Regarding set bounds, definite elements of *X* that cannot be removed (not part of *Y*) must be included in *Z*:

$$(X : \text{changed glb or Y : changed lub}) \quad \frac{a = a_z \cup \left(a_x \backslash b_y\right)}{\{Z = X\backslash Y\} \mapsto \{Z = X\backslash Y, Z \in [a, b_z]\}} \tag{54}$$

Conversely, definite elements of *X* that cannot be part of *Z* must be included in *Y*, so that they can be removed:

$$(X : \text{changed glb or Z : changed lub}) \quad \frac{a = a_y \cup (a_x \backslash b_z)}{\{Z = X\backslash Y\} \mapsto \{Z = X\backslash Y, Y \in \left[a, b_y\right]\}} \tag{55}$$

If both arguments are the same set, their difference is empty:

$$(X \text{ or } Y : \text{bound}) \quad \frac{}{\{Z = X\backslash Y, X = Y\} \mapsto \{X = Y, Z = \varnothing)\}} \tag{56}$$

If *X*=*Z*, then we can remove the difference constraint, since we have already constrained *Y* and *Z* to be disjoint:

$$(X \text{ or } Y : \text{bound}) \quad \frac{}{\{Z = X\backslash Y, X = Z\} \mapsto \{X = Z\}} \tag{57}$$

If arguments are disjoint, then the difference is the first set:

$$(X \text{ or } Y : \text{bound}) \quad \frac{(\text{ground}(X) \vee \text{ground}(Y)), X \cap Y = \varnothing}{\{Z = X\backslash Y\} \mapsto \{Z = X\}} \tag{58}$$

If both arguments are ground, we can remove the constraint:

$$(X \text{ or } Y : \text{bound}) \quad \frac{\text{ground}(X), \text{ground}(Y), z = X\backslash Y}{\{Z = X\backslash Y\} \mapsto \{Z = z\}} \tag{59}$$

Elements of *X* must be present either in *Y* or in *Z*, since if such an element is not in *Y* then it is not removed and is forcefully in difference *Z*. Consequently; the universe of *Y* and *Z* limits *X*. As a costly operation (involving the three *lub*s), it is performed only once, on instantiation of *Z*:

$$(Z : \text{ground}) \quad \frac{b = \left(b_x \cap \left(b_y \cup b_z\right)\right)}{\{Z = X\backslash Y\} \mapsto \{Z = X\backslash Y, X \in [a_x, b]\}} \tag{60}$$

Inferences on set bounds and corresponding rules of the set difference constraint are depicted in Table 3, similarly to the previous two sections.

Inclusion and disjointness constraints are posted by rule (50) (when set difference constraint is told).

Constraints involving cardinalities are:

$$\begin{aligned} C_z &\geq C_x - C_y \\ C_z &\leq C_x - \#(a_X \cap a_Y) \\ C_z &\leq \#(b_X \cup b_Y) - C_y \end{aligned}$$

corresponding to rules (50), (52), and (53).

2.2.11 Generalisation to Sets Functions

Inference capabilities over set functions, such as the cardinality, are particularly important when solving set problems where they play a special role. Other possible graded functions are the minimum and maximum of sets of numbers. Constraints over such functions

Table 3 Set difference: cause–effect rules on set bounds

Effect Change	a_x	b_x	a_y	b_y	a_z	b_z	
a_x			$a_x \backslash b_z$	–	$a_x \backslash b_y$	–	55, 54
b_x			–	–	–	b_x	$Z \subseteq X$
a_y	–	–			–	$-(a_y)$	$Y \$ Z$
b_y	–	$b_y \cup b_z$			$a_x \backslash b_y$	–	60, 54
a_z	a_z	–	–	$-(a_z)$			$Z \subseteq X$, $Y \$ Z$
b_z	–	$b_y \cup b_z$	$a_x \backslash b_z$	–			60, 55
	$Z \subseteq X$	60	55	$Y \$ Z$	54	$Z \subseteq X$, $Y \$ Z$	**Rules**

considering integer ranges for these minima or maxima can also be very useful since problems with such constraints are very common. Of course, these functions may be applied to sets of elements of arbitrary type, not just numbers, as long as a valid ordering function is associated (like Prolog's lexicographic @</2 for two arbitrary terms). Therefore, there is no reason to consider only functions mapping sets to integers. An interesting function is the sets union mapping sets of sets to sets. This function maps set $\{\{a, b\},\{a, f\},\{b, c\},\{g\}\}$ into set $\{a, b, c, f, g\}$ and satisfies (for two sets, s_1, s_2, of sets) $s_1 \subseteq s_2 \Rightarrow \bigcup s_1 \subseteq \bigcup s_2$.

In *Cardinal*, a set variable X may be represented by $[\text{in}_X + \text{poss}_X] - \text{Functions}(X)$ where in_X is its greatest lower bound (i.e. the elements known to belong to X), poss_X the set of extra elements still possible in X (poss_X and in_X are disjoint and their union constitutes the lowest upper bound of X, i.e. the elements not excluded from X), and Functions*(X)* is a set of functions of X. Each element of Functions*(X)* is a pair $f{:}V$ where f is the function identifier, and V a domain variable representing its value *f(X)*. The four currently possible functions are cardinality (#), union ($\cup$), minimum (*minimum*) and maximum (*maximum*) functions (where these last two functions apply to sets of integers).

For example, $[\{a,g\} + \{b,c,x,y\}] - \{\# : C\}$ where C is a finite domain variable with domain $\{3, 5\}$, represents a set variable with three or five elements, being two of them a and g and the rest coming from only $\{b, c, x, y\}$. A set may have more than one function attached, e.g. $[\{3,4\} + \{1,5,6,9\}] - \{\# : 4, \text{minimum} : \text{M}\}$ represents a set variable with four integers, where M will be constrained to domain $\{1, 3\}$.

The cardinality and minimum and maximum functions are integer variables, whereas the union function can be a normal *Cardinal* set variable with associated functions itself (e.g. $[\{\{b,d\}\}+\{\{a,b,e\},\{c,e\},\{a,c,d\}\}]-\{\cup : \{a,b,c,d,e\}\}$ or $[\{\{5\},\{3,5\}\}+\{\{2\},\{3\},\{2,5,6\},\{3,4\},\{1,7,8\}\}] - \{\cup : S\}$, where S can be constrained in its elements, cardinality, and even its minimum and maximum).

2.3 Implementation Notes, Complexity, and Consistency

We used ECLiPSe with attributed variables to implement *Cardinal*, the set constraint solver with cardinality inferences based on the above rules. The attributes of a set variable are its domain and its functions together with lists of suspended goals, since we used the underlying predicate suspension handling mechanism. Note that ECLiPSe provides waking conditions such as a change in some domain, but the waking constraint does not know what exactly has changed (it may only know that it has changed somehow), which is a possible source of inefficiency. For instance, with constraint $X \subseteq Y$, if element a is inserted in X, we know immediately that a must also be inserted in Y. Unfortunately, with ECLiPSe mechanisms we have to wake the constraint due to a change in X's *glb*, and then the constraint must include the whole X's *glb* in Y, since it does not know what are the new elements. This can be overcome with reactive changes as in [32]. Notwithstanding such limitations, good results were still achieved.

The cardinality of a set is an integer variable to be handled by the ECLiPSe finite domains library. To represent the domain of set S as a set interval we need its bounds a_S and b_S. Since $a_S \subseteq b_S$, it is sufficient to store a_S as the definite elements of S, and the difference $b_S \backslash a_S$ as the possible extra elements of S (both implemented as sorted lists). For efficiency reasons, the sizes of its two bounds are also stored.

The time complexity of the presented rules depends on the data structures used. With bounds' sizes stored, this complexity basically depends on the sizes of the involved sets in the operations described in the pre-conditions of each rule. In fact, in the worst case these sets (either a set bound or a ground set) must be fully traversed (i.e. in its list representation) element by element. (Of course, the use of sorted lists allows us to skip some subsets, in practice, although other representations could improve this even further.) Thus, each rule's complexity is simply given by $O(\sum n_i)$, where n_i is the size of the ith set involved in set operations in its pre-condition (e.g. a_X, b_Y, X), for all such sets. As exceptions, we have the special cardinality rules (30), (41), and (51), with no pre-conditions, where the necessary computation is inferred by the action of updating the cardinality domain of Z (result of the set intersection, union, or difference of sets X and Y), as shown in the rule's Constraint Store changes, when telling the constraint. In these cases, in addition to the four set bounds used (for X and Y), all possible cardinality pairs are checked, in the worst case, thus making complexity $O(\#a_X + \#a_Y + \#b_X + \#b_Y + D_X * \#D_Y)$.

Rules are monotonic since domains can only get smaller, and they are only triggered on changes, thus assuring reaching a fixed point (rules are also idempotent).

In general, constraints perform all the possible inferences when posted (interval-consistency on the sets; arc-consistency on their cardinalities), while their subsequent maintenance only ensures arc-consistency on their bounds. The rationale for this is that it is worth spending more time trying to reduce domains, only if this effort is not done too frequently.

2.3.1 Set Labelling

Cardinal, as the majority of constraint solvers, is not complete, which means that even when constraint propagation is successful, CSP variables must still be instantiated in order to prove that a solution is possible (or not). Since set variables are represented by set intervals, we can split the search space in two (similarly to the usual way of handling intervals over reals) by a disjunction on the membership of the set's possible extra elements. I.e. $x \in b_S \backslash a_S$ either belongs or not to set S. Hence, at each such try (disjunction solving), the set domain is restricted either by adding x to a_S or by removing it from b_S. This allows making a more active use of constraints during the search phase, avoiding instantiating the set directly to some element in the domain. Such direct instantiation of set variable S to a particular ground set s can be hard to succeed for large domains (usually the case, with set intervals), and if it fails, then only another constraint, $S \neq s$, is derived, which hardly will restrict the domain of S (or of any other variable). This naïve labelling, generally, only succeeds after many failures. *Conjunto* implements set labelling as a recursive refine procedure over disjunctions of the form (x in S ; x notin S) for possible elements x of set S. This means that inclusion of x is always tried first. However, we realised that often a labelling strategy of first trying exclusion is more effective. Hence, in *Cardinal*, we implemented set labelling with an extra parameter (*up* or *down*) that indicates what choice to try first. When given value 'up,' a set is labelled as in *Conjunto*, while for value 'down,' choices are handled as (x notin S ; x in S), which is actually the default in Cardinal.

3 Modelling Digital Circuits Problems

A digital circuit is composed of gates performing the usual Boolean logic operations (*and*, *xor*, *not*, ...) on their input bits to determine the output bits. A digital signal has two possible values, 0 or 1, and the circuit gates (or their connections) might be faulty. We will only address the usual *stuck-at-X* faults (*X*=0 or 1), whereby the output of a gate is *X* regardless of its input.

For some circuit under consideration, let n_i and n_o be the number of input and output bits, respectively, *I* is the set of all possible inputs ($\#I=2^{ni}$), *out(b, i, F)* the output value for bit number *b* ($b\in 1..n_\mathrm{o}$) under input *i* ($i\in I$) when the circuit exhibits a set of faults *F*. With such notation, a number of Test Generation (TG) related problems can be formulated:

1. TG (basic). Find an input test pattern *i* for a set of faults *F*, i.e. an input for which some output bit of the circuit is different when the circuit has faults *F* or has no faults.

$$\mathrm{test}(F,i) \Leftrightarrow \exists b \in 1..n_o, \mathrm{out}(b,i,F) \neq \mathrm{out}(b,i,\varnothing)$$

2. Differential Diagnosis. Find an input test pattern *i* that differentiates two diagnostic sets *F* and *G*, i.e. an input *i* for which some output bit of the circuit is different when the circuit has faults *F* or *G*

$$\mathrm{diff}(\{F,G\},i) \Leftrightarrow \exists b \in 1..n_o, \mathrm{out}(b,i,F) \neq \mathrm{out}(b,i,G)$$

   This problem (2) is also listed in CSPlib at http://www.csplib.org [19] as problem number (43).

   Let *D* now denote a set of diagnostic sets (these can be a set of more common faults, but in the limit *D* may represent all possible sets of faults in the circuit). The next two related optimisation problems deal with sets with varying cardinality.

3. Maximisation. Find an input *i* which is a test pattern for the maximum number of diagnoses in *D*. Set $D_i \subseteq D$ now denotes the set of faulty gates for which input *i* is an input test pattern, i.e. $D_\mathrm{i} = \{F : F \in D \wedge \mathrm{test}(F,i)\}$

$$\max(D,i) \Leftrightarrow \forall j \in I, \#D_j \leq \#D_\mathrm{i}$$

4. Minimisation. Find a minimal set *S* of input test patterns that cover all diagnoses in *D* (the definition of covering is given below, where *P(I)* is the power-set of *I*).

$$\begin{aligned}\mathrm{cover}(D,S) &\Leftrightarrow \forall F \in D, \exists i \in S : \mathit{test}(F,i)\\ \min(D,S) &\Leftrightarrow \mathrm{cover}(D,S) \wedge \forall S' \in P(I), \#S' \geq \#S \vee \neg\mathrm{cover}(D,S')\end{aligned}$$

Given these problems, we now present two ways to model them: (1) representing digital signals with sets and Booleans, and (2) adopting a pure set representation.

3.1 Modelling Digital Signals with Sets and Booleans

Since the faulty behaviour can be explained by several of the possible faults, we represent a signal not only by its normal value but also by the set of diagnoses it depends on. More specifically, a signal is denoted by a pair *N–L*, where *N* is a Boolean value (representing the Boolean value of the circuit if it had no faults) and *L* is a set of diagnostic sets, that might change the signal into the opposite value. For instance, $X = 0 - \{\{f/0, g/0\}, \{i/1\}$ 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 $N-\varnothing$ represents a signal with constant value *N*, independent of any fault.

Any circuit gate either belongs to the universe D of possible faults or not. We next show how to model the different gate types in order to process signals in the form of pairs N–L.

3.1.1 Normal Gates

Normal gates (those with no faults included in D) fully respect the Boolean operation they represent. We discuss the behaviour of *not*-and *and*-gates as illustrative of such gates; the others can be modelled as combinations of these.

Given the above explanation of the encoding of digital signals, it is easy to see that, for a normal *not*-gate whose input is signal N–L, the output is simply $\overline{N}$–L, since the set of faults on which it depends is the same as for the input signal.

As for the *and*-gate, three distinct situations may arise as illustrated below:

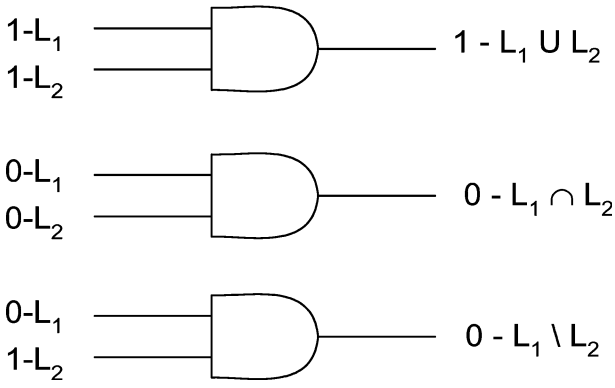


Fig. 6 And-gate

In the absence of faults, the output is the conjunction of the normal inputs. However, the output can be different from this normal value due to faulty inputs. In the first case of Fig. 6, with two 1s as normal inputs, it is sufficient that a fault in either set L_1 or L_2 occurs for the output to change, thus justifying the disjunction of the sets in the output signal. In the second case (two 0s), it is necessary that faults occur in both L_1 and L_2 to invert the output signal, thus imposing an intersection of the input sets. In the last case, to obtain an output different from the normal 0 value, it is necessary to invert the normal 0 input (i.e. to have faults in set L_1) but not the normal 1 input (i.e. no faults in set L_2) which justifies the set difference in the output.

3.1.2 S-buffers

The gates that participate in the universe of faults D (i.e. that can either be *stuck-at*-0 or *stuck-at*-1) may be modelled by means of a normal gate to which a special buffer, an *S-buffer*, is attached to the output. As such, all gates are considered normal, and only S-buffers can be stuck. An S-buffer for a gate g has associated to it a set L_S of diagnostic sets where g appears as stuck. Since g can appear either as *stuck-at*-0 or *stuck to at*-1, we split this set in two (L_{S0} and L_{S1}), one for each type of diagnoses:

$$\begin{aligned} L_{S0} &= \{\text{diag} \in D : g/0 \in \text{diag}\} \\ L_{S1} &= \{\text{diag} \in D : g/1 \in \text{diag}\} \\ L_S &= L_{S0} \cup L_{S1} \end{aligned}$$

Table 4 S-buffer output

In	Out
$0{-}\varnothing$	$0{-}L_{S1}$
$1{-}\varnothing$	$1{-}L_{S0}$
$0{-}L_i$	$0 - L_{S1} \cup (L_i \setminus L_{S0})$
$1{-}L_i$	$1 - L_{s0} \cup (L_i \setminus L_{S1})$

The modelling of S-buffers is shown in Table 4. When the input is 0 independently of any fault, the S-buffer output would normally be also 0, but if it is *stuck-at*-1 then it becomes 1, thus depending on set L_{S1}. More generally, if the normal input is 0 but dependent on L_i, the output depends not only on L_{S1} but also on input dependencies L_i (except if they include fault *g*/0). The same reasoning can be applied to the case where the normal input signal is 1, and the whole Table 4 is generalised as shown in Fig. 7.

3.1.3 Modelling the Problems

To model the diagnosis problem, and differentiate faults in set F from faults in set G, (the universe is thus $D = \{F,G\}$ of cardinality 2), either $N-\{F\}$ or $N-\{G\}$ must be present in a circuit output bit. In any case, a bit $N–L$ must be present in the output with $\#L=1$.

The goal of the maximisation problem is to maximise the number of output dependencies, i.e. the number of diagnoses covered by the input test pattern. The goal is then maximise $\#(\cup_b L_b)$ where b ranges over all the output bits b with signals $N_b–L_b$.

The fourth problem (minimisation) is a typical set covering problem: the test patterns (i) are the resources, and the diagnoses (F) are the services we want to cover with minimum resources. Each diagnosis can be tested by a number of test patterns, and each test pattern can test a number of diagnoses. The relation between these services and resources is *test(F,i)*, which is not fully known a priori, though.

3.2 Modelling Digital Signals with Sets

With the previous representation, all digital signals are represented by a pair: a Boolean value that the signal takes if there were no faults at all, plus a set of faults on which it depends. Both the Boolean value and the set can be variables, enforcing constraints on two domains to be expressed for each gate, and the modelling presented above implies an extensive use of disjunctive constraints, with the corresponding exponential complexity. For instance, to express the above *and*-gates, one needs to know the Boolean values of the signals to select the appropriate sets constraint.

It would thus be very convenient to join the two domains into a single one. Intuitively, to incorporate the two domains, the new one should be richer than any of them. But there is also the possibility of using a simpler one if the loss of information is not important for the problem, or if it can be compensated by the introduction of extra constraints. This latter alternative is the one we follow here.

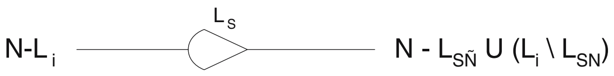


Fig. 7 S-buffer

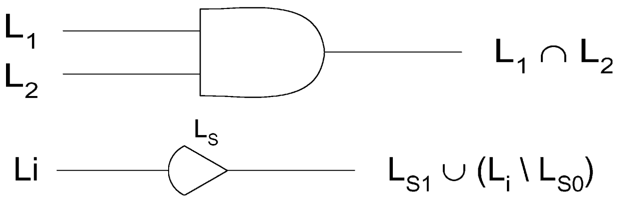


Fig. 8 And-gate and S-buffer over sets

More specifically, we propose the use of a transformation *transf*, where signals $0-L$ are simply represented as L, and $1-L$ as $\overline{L}$ (the complement of L, w.r.t D):

$$\text{transf}(S) = \begin{cases} L, & S = 0 - L \\ \overline{L}, & S = 1 - L \end{cases}$$

Although *transf* is not a bijective function (both $0-L$ and $1-\overline{L}$ are transformed into L*)*, we argue that it is quite useful to model our problems. For example, with this representation, the *and*-gate and the S-buffer are simply stated as follows: stated as in Fig. 8.

The correctness of this new simplified representation can be checked by simple analysis of each case, shown in Tables 5 and 6.

In Table 5, the transformed output set is always the intersection of the transformed input sets, i.e. $\text{transf}(I1) \wedge \text{transf}(I2) = \text{transf}(I1 \wedge I2)$. Similarly, in Table 6, s_buffer(L_S,transf (Input))= transf(s_buffer(L_S, Input)).

For completion, it may also be noticed that the other gate operations can be expressed with the expected set operations (see Fig. 9).

3.2.1 Modelling the Problems

To solve the diagnosis problem with this representation based exclusively on sets, it is still sufficient to ensure that a set L with cardinality 1 is present in an output bit of the circuit. Being $D=\{F, G\}$ the set of diagnoses F and G to differentiate, it is equivalent to have in an output bit an L ($\#L$=1), in the sets representation, or to have an N–L ($\#L$=1) in the mixed representation (pairs set-Boolean).

Proof

$\Rightarrow$ If L ($\#L$=1) is present in an output bit, it represents either $0-L$ ($\#L$=1) or $1-\overline{L}$ ($\#\overline{L} = 1$, since $\#D$=2). In either case, it solves the problem.

$\Leftarrow$ If an N–L ($\#L$=1) is present in an output bit, it is either $0-L$ (represented as L) or $1-L$ (represented as $\overline{L}$). In either case, the represented set has cardinality 1. ■

Therefore, the loss of information incurred by the transformation used has no effect on this problem, since it is not necessary to add any new constraints to solve it.

Modelling the maximisation problem in a circuit c, is not so straightforward. Since a digital signal coded as D does not necessarily mean a dependency on all diagnoses (it can represent 0-D, as well as $1-\emptyset$), maximising the union of all the output bits is not adequate.

Table 5 Application of transf function to the inputs and output of an and-gate

$I1$	transf($I1$)	$I2$	transf($I2$)	$I1 \wedge I2$	$\text{transf}(I1 \wedge I2)$
$1-L_1$	$\overline{L_1}$	$1-L_2$	$\overline{L_2}$	$L_1 \cup L_2 - 1$	$\overline{L_1 \cup L_2} = \overline{L_1} \cap \overline{L_2}$
$0-L_1$	L_1	$0-L_2$	L_2	$L_1 \cap L_2 - 0$	$L_1 \cap L_2$
$0-L_1$	L_1	$1-L_2$	$\overline{L_2}$	$L_1 \setminus L_2 - 0$	$L_1 \setminus L_2 = L_1 \cap \overline{L_2}$

Table 6 Application of transf function to the input and output of an S-buffer

In	transf(In)	S-buffer output	transf(output)
$0{-}L_{\mathrm{i}}$	L_{i}	$0 - L_{S1} \cup (L_i \backslash L_{S0}))$	$L_{S1} \cup (L_i \backslash L_{S0})$
$1{-}L_{\mathrm{i}}$	$\overline{L_i}$	$1 - L_{S0} \cup (L_i \backslash L_{S1})$	$\overline{L_{S0} \cup (L_i \backslash L_{S1})} = \overline{L_{S0}} \cap \overline{L_i \cap \overline{L_{S1}}} = \overline{L_{S0}} \cap \left(\overline{L_i} \cup L_{S1}\right) = \left(\overline{L_{S0}} \cap \overline{L_i}\right) \cup \left(\overline{L_{S0}} \cap L_{S1}\right) = \left(\overline{L_i} \cap \overline{L_{S0}}\right) \cup L_{S1} = L_{S1} \cup \left(\overline{L_i} \backslash L_{S0}\right)$

In fact, it is necessary to know exactly whether an output signal depends on its set or on its complement. This can be done as shown in Fig. 10.

Circuit *c* with S-buffers is kept as before, but now the circuit with no S-buffers is added, sharing the input bits and with the corresponding output bits xor-ed. Values inside the normal circuit are necessarily independent of any faults, and can only be represented as ∅ (for 0−∅) and *D* (for 1−∅). The *xor*-gates in the output bits, which receive a set *L* from the faulty circuit and either ∅ or *D* (the universe) from the normal one, recover the correct dependency set of the signal as being either *L* or $\overline{L}$ (if the normal value were 0 or 1, respectively). A maximisation on the union of these real fault dependencies can then be performed to obtain the desired solution to the problem.

The reduction of the problem size by eliminating the Boolean part of the domain is now compensated by the duplication of constraints. Still, what could naively be seen as a useless manipulation, allows an active use of constraints by a set constraint solver avoiding the choice-points that would otherwise be necessary.

Moreover, the exponential component of search, labelling, is only performed at the circuit with S-buffers (the other circuit simply checks this labelling). This is in contrast with Boolean SAT approaches, which consider one extra circuit for each diagnosis, which is unacceptable, in practice, for a large set of diagnoses [31].

The minimisation problem is a meta-problem: it involves sets of solutions to set problems. A set variable *S* could be used ranging from ∅ to *P(I)*, where set *S* of inputs is constrained to cover diagnoses *D*. The goal is then to minimise the cardinality of *S*.

To find a test pattern for a single diagnosis *F* using sets, we need to model a faulty as well as a normal circuit, xor-ing the outputs and checking whether at least one set value {*F*} is obtained. This is equivalent to the SAT approach for obtaining test patterns.

The ideal is to consider all diagnoses *D* simultaneously, with set constraints, and include or remove elements from *S* during the computation, updating the covered diagnoses until *D* is reached, and then start finding smaller sets for *S* in a branch-and-bound manner. This is still an open problem and perhaps the maximisation problem can be used to solve this minimisation one, by obtaining intermediate solutions.

4 Other Problems and Results

In this section we present results for the differentiation problem and we discuss other applications for *Cardinal*. Set covering is also discussed in Section 4.3 when analysing *Cardinal* extensions.

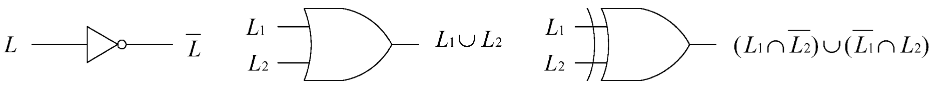
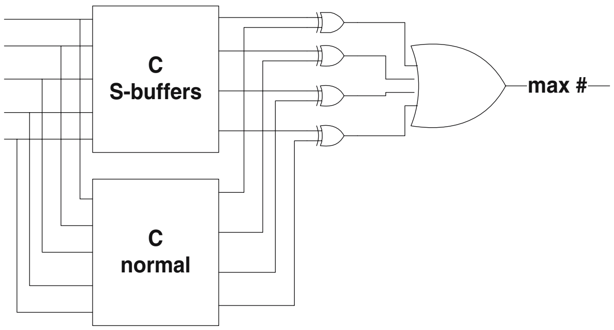


Fig. 9 Other gates over sets

Fig. 10 Modelling the maximisation problem with sets



4.1 Differential Diagnosis

Given the transformation presented in the previous section to encode circuit signals using sets, independent values such as the circuit inputs are represented by $[\varnothing, U]_{Cx:\{0,u\}}$. For the discussed circuit problems, the universe is the set of diagnoses D.

In this section we focus on the sets model for the differential diagnosis problem, which we applied to the standard ISCAS digital circuits benchmarks [26] (circuits' statistics are summarized in Table 7).

From a set of differentiation benchmarks created over these circuits [3], we randomly picked pairs of diagnoses to differentiate.

Gate constraints are implemented based on basic set operations. Exclusive set union (*xor*-gate) is not currently implemented as a basic operation, hence it is defined as "$S_1 \oplus S_2$" $\equiv \left(S_1 \cap \overline{S_2}\right) \cup \left(\overline{S_1} \cap S_2\right)$.

After initial constraint propagation, there is a subset of POs that can have cardinality 1 (the differentiation goal). Our labelling strategy tries these relevant output bits in an iterative time-bounded search (ITBS [5]). Then, for such a PO and its path to S-buffers, we label the PIs they depend on, by assigning values (0 or u) to their cardinality. If successful, the rest of the circuit input is labelled.

In Table 8 we present results for different set libraries over ECLiPSe with Linux on a Pentium II/450 Mhz, obtained during a stay in IC-Parc in cooperation with Joachim Schimpf, Carmen Gervet and Mark Wallace. *Cardinal* is tested together with *Conjunto* and *fd_sets* (an optimised set library of ECLiPSe for integer sets, by the ECLiPSe team and Neng-Fa Zhou).

We are trying to differentiate two diagnoses (*Diag*1 and *Diag*2) over the ISCAS circuits. Such a pair of diagnoses (each consisting of one or two stuck gates) can be differentiable (marked with √) or not (marked with X). Results are in seconds and 'na' values mean non-available results (i.e. aborted execution after hours of processing). For example, the first line reports that the differentiation of gate 380*gat stuck-at*-0 from gate 415*gat stuck-at*-1, in circuit *c*432, took 12 seconds in *Conjunto* and 0.4 seconds in *Cardinal* (time to prove it is

Table 7 ISCAS circuits: number of Primary Inputs (PI), Primary Outputs (PO), and Gates (G)

	*c*432	*c*499	*c*880	*c*1355	*c*1908	*c*2670	*c*3540	*c*5315	*c*6288	*c*7552
PI	36	41	60	41	33	233	50	178	32	207
PO	7	32	26	32	25	140	22	123	32	108
G	160	202	383	546	880	1,193	1,669	2,307	2,416	3,512

Table 8 Differentiation results over different set libraries

Circuit	Diag1	Diag2	Diff.	Conjunto	fd_sets	Cardinal
*c*432	380gat/0	415gat/1	X	12.0	7.3	0.4
	431gat/0	428gat/1	√	**7.0**	4.4	0.4
	431gat/0	419gat/1	√	7.0	4.4	0.4
	428gat/1	419gat/1	√	9.7	6.1	0.5
	431gat/0	419gat/0	√	6.9	4.4	0.4
	428gat/1	419gat/0	X	0.8	6.3	0.3
*c*499	od2/0	id2/0	√	0.2	6.1	2.5
	od2/0, od19/0	id2/0, od19/0	√	0.3	6.1	2.5
*c*880	389gat/1	291gat/1	X	1.6	11.4	0.5
	422gat/0, 850gat/0	422gat/0, 840gat/0	X	0.1	1.2	0.5
*c*1355	1258gat/1	1339gat/0	√	**688.2**	189.9	1.0
	162gat/0	1274gat/1	√	**968.0**	0.7	1.1
	1347gat/0	1315gat/0	√	686.0	391.5	0.8
*c*1908	1541/1	1538/0	√	14.3	11.2	1.3
	860/1	72/1	√	251.9	185.7	1.3
	72/1	71/0	X	315.6	232.8	1.1
*c*2670	96/1	221/0	√	0.5	1.2	1.7
	217/0	216/1	X	37.3	22.3	1.4
	162/1	1467/0	√	1.1	1.5	1.8
	236/0	120/1	√	0.5	1.2	1.7
*c*3540	855/0	707/1	√	0.5	1.4	2.2
	955/0	954/0	X	5579.1	3366.0	1.9
	403/0	3544/1	√	na	335.3	2.3
*c*5315	91/0	742/0	√	6.6	5.9	3.5
	651/1	649/1	√	7.7	6.6	3.5
	649/1	1598/1	X	296.4	188.6	2.8
*c*6288	5671gat/0	5537gat/1	√	na	na	4.7
	6288gat/1	6285gat/0	√	na	na	4.6
	813gat/0	6123gat/0	√	na	na	4.6
*c*7552	5222/1	5221/1	X	16.9	150.9	5.6
	4772/1	330/1	√	14.3	12.2	4.8

impossible, in this case). *Conjunto* results are not very reliable since this solver was still 'buggy' at the time (bold values represent tests where an incorrect solution was detected). Nevertheless, we think that these values provide us with an overall idea of the mean execution time of *Conjunto* over these problems.

As can be seen in the table, *Cardinal* is always able to find the solution in 0.4–5.6 secods, taking 1 minute to solve the whole set of 31 tests, while even *fd_sets* could not solve any differentiation for *c*6288 and for other circuits it can take 2 or 5 minutes or even an hour to solve a single test.

To compare the performance of our set model with one using an eight-valued logic [5], we ran the same set of differentiation tests, this time on a Pentium 4, 1.7 GHz, 512 Mb RAM, using ECLiPSe Prolog 5.1 (Tcl/Tk version) under Windows 2000.

Although generally faster, the eight-valued logic model required about 74 seconds to solve all tests, while *Cardinal* required only 23 seconds. This was due to a single test in circuit *c*6288 which took 1 minute solving, while *Cardinal* never needed more than 2 seconds for each test.

To assess the advantages of cardinality inferences, we took off inferences from *Cardinal* that are not implemented in *Conjunto* (i.e. some rewrite rules regarding cardinality and other especial inferences). This simpler version is referred to as "Conjunto". We then tried to solve the same problem using *Cardinal* and "Conjunto". Results are shown in Table 9 (times in seconds on a Pentium III, 500 Mhz).

Globally, it can be stated that *Cardinal* showed a speed-up of two orders of magnitude compared to "Conjunto" on this set of problems (and others we tried) although, as expected, the improvement was not uniform over all the tests.

While for circuit *c*432 *Cardinal* showed a consistent speed-up around 25, for larger circuits the variation can be quite large. In circuits *c*1908 and *c*3540 the speed-up ranges from 0.6 to 1,451.2, with *Cardinal* being more efficient on harder problems (specially those where there is no differentiating pattern between the two diagnoses). For the two instances in circuit *c*3540 where there was an easy solution for "Conjunto," *Cardinal* was slower due to the extra inferences performed, and the times thus reflect this overhead. The extra computing effort may be largely compensated, as tests in *c*6288 show, where *Cardinal* easily found a solution, whereas "Conjunto" had to be aborted in all three tests after 1 hour (one particular test was even kept running for 1 day of unsuccessful processing). Of course, the speed-up can be arbitrarily large as long as not enough propagation was achieved and we start labelling variables, since the execution time is exponential on the number of these variables.

Due to all the especial inferences and list processing, we expected *Cardinal* to experience problems with larger circuits or in problems with many diagnoses. Also, since the general feeling is that, in practice, it is very costly to perform all the desired inferences over sets and their cardinalities, we tried to create another version with n-ary gates but with fewer

Table 9 Experimental results

Circuit	Diag1	Diag2	Diff.	"Conjunto"	Cardinal	Speed-up
*c*432	380gat/0	415gat/1	X	8.1	0.4	20.3
	431gat/0	428gat/1	√	39.4	1.3	30.3
	431gat/0	419gat/0	√	37.9	1.4	27.1
	428gat/1	419gat/0	X	24.0	1.0	24.0
*c*1908	1541/1	1538/0	√	24.2	3.2	7.6
	860/1	72/1	√	156.3	1.3	120.2
	72/1	71/0	X	194.9	1.0	194.9
*c*3540	855/0	707/1	√	4.1	6.1	0.7
	955/0	954/0	X	3482.8	2.4	1451.2
	855/0	707/1	√	1.8	3.2	0.6
	403/0	3544/1	√	352.1	2.9	121.4
*c*6288	5671gat/0	5537gat/1	√	>86400	11.0	>7854.5
	6288gat/1	6285gat/0	√	>3600	8.9	>404.5
	813gat/0	6123gat/0	√	>3600	8.9	>404.5

Table 10 Differentiation results with different solvers

	N	NC	ND	8V	Sets
*b*432_*a*_1	4	1	3	0.09	0.33
*b*432_*a*_2	6	5	11	44.00	3.78
*b*432_*b*_1*a*	3	2	2	0.08	0.30
*b*432_*b*_1*b*	2	2	1	0.06	0.17
*b*432_*b*_2	4	4	6	0.30	3.02
*b*432_*c*_1	2	2	1	0.03	0.19
*b*7552_*a*_1*a*	13	9	43	55.38	101.02
*b*7552_*a*_1*b*	13	9	43	53.86	112.63
*b*7552_*b*_1	13	12	73	223.75	179.47
*b*7552_*c*_1*a*	16	6	25	22.87	49.24
*b*7552_*c*_1*b*	14	7	29	30.12	61.95

inferences, which produced results that were midway between "Conjunto" and *Cardinal*. The fact is that *Cardinal* still managed to efficiently solve problems for the largest of the benchmark circuits (*c*7552) so no improvements were obtained by reducing inferences.

For a more exhaustive comparison between different models, we picked the smallest and largest ISCAS circuits, namely *c*432 and *c*7552 and executed the respective differentiation benchmarks (partitioning diagnoses into sets of equivalence classes) described in [3], to obtain the results of Table 10, where total time for each benchmark is shown in seconds for both *Cardinal* (CLP(*Sets*)) and a model using the 8-valued logic (CLP(*FD*)).

N denotes the number of initial diagnoses; *NC* is the number of computed classes and *ND* is the number of differentiation tests required to obtain the final partition. The trend for the eight-valued logic solver (denoted as 8V) to be faster than using sets is confirmed in general but, interestingly, harder instances take longer time to solve. In fact, for *c*7552, 8V solves four benchmarks about twice as fast but requires much more time for *b*7552_*b*_1, for which *Cardinal* is faster. Also, with *c*432, 8V required 44 seconds to solve *b*432_*a*_2, while *Cardinal* solved it in less than 4 seconds.

Thus, it seems that the extra complexity of set solving, although heavier for easier problems, produces more consistent results by being less dependent on phase transitions [12].

4.2 Warehouse

In this section we show a simple example of a traditional application with straightforward set modelling. In fact, generally we do not need special data transformations, as described for digital circuits, to model problems with sets and solve them efficiently. Often, problems are already naturally specified with sets, as exemplified in the next sections.

The warehouse problem is an optimisation problem consisting in deciding which warehouses to build from a set of known locations, so that a given set of customers is served with the minimum cost. There is a cost per customer being delivered from a specific warehouse, whereas the cost of building a new warehouse is constant and unique: 50,000.

A particular instance of this problem, with 20 customers and 19 warehouses, is shown in Table 11, with the optimum solution stressed (built warehouses and deliveries in bold).

An FD/ILP model of this problem uses 19 0–1 variables W_i to express whether warehouse i ($i \in \{1..19\}$) is built or not. Twenty customers variables C_j take $\{1..19\}$ as integer domain, meaning that customer j ($j \in \{1..20\}$) is served by warehouse C_j with cost

Table 11 Delivery costs of warehouse problem

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Roissy	68,948	**15,724**	**24,300**	**4,852**	40,950	66,330	39,698	45,895	**5,519**	53,433	47,235	**13,125**	**138,176**	**106,993**	**55,408**	**55,786**	**16,847**	**27,780**	**4,278**	**9,672**
Orly	68,948	8,634	24,300	4,852	40,950	66,330	39,698	23,975	4,387	53,433	47,235	13,125	159,783	123,723	47,915	93,864	16,847	24,308	3,983	9,005
Lille	68,948	17,850	12,600	4,852	78,390	66,330	39,698	45,895	9,481	53,433	47,235	14,175	181,390	140,454	62,900	106,556	19,125	31,253	4,573	12,340
Nancy	68,948	17,850	24,300	2,817	31,590	66,330	39,698	45,895	9,481	53,433	47,235	18,375	181,390	140,454	62,900	106,556	19,125	31,253	4,573	12,340
Lyon	35,101	23,520	60,300	4,852	16,380	34,650	15,998	23,975	9,481	21,533	19,035	18,375	239,008	185,069	82,880	140,403	25,200	31,253	4,573	9,672
Rouen	68,948	16,433	24,300	6,104	40,950	66,330	39,698	45,895	4,387	53,433	47,235	7,350	166,985	129,300	57,905	98,095	17,607	28,938	4,573	9,672
Toulouse	24,524	46,200	60,300	10,486	40,950	13,860	14,813	18,495	5,519	21,533	24,675	35,175	469,480	363,528	162,800	275,792	49,500	77,553	5,753	12,340
Montpellier	**24,524**	46,200	60,300	10,486	**31,590**	**24,750**	**8,295**	**23,975**	9,481	**19,938**	**19,035**	35,175	469,480	363,528	162,800	275,792	49,500	77,553	5,753	12,340
Bordeaux	35,101	23,520	60,300	10,486	40,950	26,730	20,738	9,590	4,387	27,913	47,235	35,175	239,008	185,069	82,880	140,403	25,200	40,513	5,753	9,672
Nantes	68,948	17,850	60,300	10,486	78,390	34,650	39,698	18,495	2,547	53,433	47,235	14,175	239,008	140,454	82,880	140,403	19,125	40,513	4,573	9,672
Marseille	26,639	46,200	60,300	10,486	31,590	26,730	14,813	23,975	9,481	11,165	17,625	35,175	469,480	363,528	162,800	275,792	49,500	77,553	9,883	22,345
Nice	35,101	46,200	60,300	10,486	31,590	34,650	15,998	45,895	9,481	19,938	9,870	35,175	469,480	363,528	162,800	275,792	49,500	77,553	9,883	22,345
Dijon	68,948	17,850	31,500	4,852	29,250	34,650	20,738	45,895	5,519	27,913	24,675	18,375	181,390	140,454	62,900	106,556	19,125	31,253	4,573	9,672
Rennes	68,948	17,850	31,500	10,486	78,390	66,330	39,698	23,975	4,104	53,433	47,235	14,175	181,390	140,454	62,900	106,556	19,125	31,253	4,573	9,672
Tours	68,948	17,850	31,500	6,104	40,950	34,650	20,738	18,495	4,387	53,433	47,235	14,175	181,390	140,454	62,900	106,556	19,125	31,253	4,278	8,671
Orleans	68,948	15,724	24,300	4,852	31,590	34,650	20,738	23,975	4,387	53,433	47,235	14,175	166,985	129,300	57,905	98,095	17,607	28,938	2,655	8,671
Chalons	68,948	16,433	24,300	4,539	40,950	66,330	39,698	45,895	5,519	53,433	47,235	14,175	166,985	129,300	57,905	98,095	19,125	28,938	4,573	9,672
Clermont	26,639	17,850	60,300	6,104	29,250	26,730	15,998	18,495	5,519	21,533	24,675	18,375	239,008	185,069	62,900	140,403	25,200	31,253	4,573	9,672
Annecy	35,101	23,520	60,300	4,852	28,080	34,650	15,998	45,895	9,481	21,533	19,035	35,175	239,008	185,069	82,880	140,403	25,200	40,513	5,753	12,340

given by *cost(j,Cj)* as the respective value of the costs table. The cost we want to minimise is then

$$\text{Cost} = 50,000 * \sum_i W_i + \sum_j \text{cost}(j, Cj)$$

A sets model uses a single set variable W_s for the built warehouses, with an empty *glb* and a *lub* consisting of all warehouses {1..19}. Customer deliveries may also be modelled as the 20 integer variables C_j of ILP. The number of warehouses in the cost function is now simply $\#W_s$ instead of a sum of 19 variables.

Implementation of these models in ECLiPSe use built-in constraints *element/*3 and *sumlist/*2 to, respectively, retrieve delivery costs and sum them. Since we want to minimise costs, we start labelling by trying to eliminate warehouses (assign 0 to W_i variables, or reduce *lub* of W_s). As soon as we decide not to build warehouse *i*, it must be removed from the domains of C_j. This is assured, for each customer *j*, by constraint *occurrences(i,Cj,*0*)* in FD, or by C_j::*lub*(W_s) in the sets model (where :: represents domain declaration). Delivery costs are labelled by picking the minimum value of the table column for each customer. The minimum cost is 730,567 with warehouses built in Roissy and Montpellier. The solution given by the sets model is W_s={1, 8} (or {Roissy, Montpellier}), which is more natural than the ILP solution for 19 variables (1000000100000000000). The program is also more declarative in CLP(*Sets*). We tested this problem for ILP, *Cardinal*, "Conjunto" and the real *Conjunto* on a Pentium III/500 Mhz, all of them reaching and proving the optimum in about 55 hundredths of a second. We first notice that set constraint programming achieved the same result as ILP. Comparing the different set solvers, for this problem where cardinality was not particularly important, *Cardinal* behaved as "Conjunto", thus showing no overhead for cardinality inferences. "Conjunto" also behaved as the real *Conjunto*. (For equivalence, we implemented the labelling strategy of refining down the warehouses in the *Conjunto* model, since it is not built-in.)

4.3 Set Covering

In Section 2.2.11 we described *Cardinal*'s extension to constraint solving over set variables with attached set functions with special inferences over them. After diagnostic problems were considered where cardinality played a special role, in this section we address set covering problems where the sets union function is used to make additional inferences thus pruning search space. We then compare experimental results with other approaches to attest the expressiveness and efficiency of *Cardinal.*

Set covering is an optimisation or satisfaction problem well studied in Operations Research (OR) that often occurs in real life. The basic goal is to find a subset of a given set of sets whose union contains all their possible elements. Let us consider a simple satisfaction example in the usual 0–1 Integer Linear Programming (ILP) approach:

$$\begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \geq \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad \text{and} \quad x_1 + x_2 + x_3 + x_4 = 2$$

where each of the four x_i is a 0–1 variable.

Intuitively, the goal is to find two columns in the 5×4 matrix, where we can find at least one value 1 on each line. The solution is then x_1=1 and x_4=1.

Given a set-covering problem, a set-based model should be quite natural. What we want to find is indeed simply a set of two columns so that all lines are "covered". One way to represent this is:

$$\begin{gathered}\text{Cols} \subseteq \{1,2,3,4\}, \quad \#\text{Cols} = 2,\\ \{1,4\} \cap \text{Cols} \neq \varnothing\\ \{1,2\} \cap \text{Cols} \neq \varnothing\\ \{3,4\} \cap \text{Cols} \neq \varnothing\\ \{2,4\} \cap \text{Cols} \neq \varnothing\\ \{1,3\} \cap \text{Cols} \neq \varnothing\end{gathered}$$

Now the solution is Cols={1, 4}.

But *Cardinal* has yet another modelling solution to this problem by means of the implemented union function of a set variable. Although the cardinality function is always present with a finite domain variable, the union function is optional since it only applies to sets of sets.

For the previous example with the union function, we can simply declare a set variable S as:

$$S :: [\{\} + \{\{1,2,5\},\{2,4\},\{3,5\},\{1,3,4,\}\}] - \{\# : 2, \cup : \{1,2,3,4,5\}\}$$

for a set with two yet unknown elements (sets representing the matrix columns), whose union cover all five integers from 1 to 5 (representing the lines). Now, labelling this single variable yields the solution S={{1, 2, 5},{1, 3, 4}}.

Our sets approach with attached functions may thus look like a global constraint. In fact, reasoning with functions of set variables can capture in a very natural manner many classes of problems, without jeopardising efficiency. For the union function this is achieved with inferences triggered by a change in the set variable bounds, such as: if set of sets X is known to include one more set s_1 (i.e. in_X grows by s_1), then its union must contain s_1, and if set s_2 is definitely excluded from X (removed from poss_X), then the elements of s_2 that were the only support for being present in the union must be removed from the union variable. Similarly, if the union function value is really given by a variable, so should the changes of its bounds affect the set variable it is attached to. The cardinality of the elements of X is also taken into account to possibly lead to early failures.

In any of the two *Cardinal* models, only one variable is needed, in contrast to ILP that requires one variable for each "column." Furthermore, with the union function of *Cardinal* only one constraint is needed to state that the universe must be covered, while ILP and the other *Cardinal* model require one constraint for each "line."

To test the different covering approaches, we turned to a digitally available OR library [7] with set covering benchmarks and we picked file *scpcyc*06, a 240×192 problem matrix corresponding to a CYC problem (number of edges required to hit every four-cycle in a hyper-cube).

The number n of columns that cover all 240 lines ranges from 0 to 192, being n=0 trivially impossible and n=192 trivially satisfied. We want to find the optimum (minimum) in between these numbers. For our test we tried to find a solution starting with n=192 and successively decrementing n until it was impossible (although faster solutions may be obtained with a Branch-and-Bound minimise predicate). Similarly, starting with n=0 we incremented it until obtaining a solution. For a given n, we imposed a time limit of a couple of hours, so that we could be left with a lower and an upper bound for the optimum.

In addition to the three proposed models (ILP plus 2 set-based ones), we included for a fair comparison a *Conjunto* version, since the first simple set-based model could also be

Table 12 Optimum bounds

	ILP Model (CLP(FD))	Conjunto	Cardinal (simple)	Cardinal (with union function)	CPLEX
Lower	9	8	8	50	54
Upper	76	80	78	78	63
Difference	67	72	70	28	9

directly applied on it. All versions were tested using CLP (on ECLiPSe), an area where we wanted to improve techniques for problem solving and which has already shown efficiency advantages in tackling a number of problems. Unfortunately, set-covering is not one of them—in fact, it is a typical problem to be solved with a specialised ILP tool (our ILP model relied on the underlying CLP(*FD*) solver to search for a solution with Boolean variables). For an idea on the difference of approaches and platforms, we also include in Table 12 the bounds obtained when trying to solve the set-covering problem with the ILP model on CPLEX [15].

First of all we notice that *Conjunto* and *Cardinal* with the same model, achieve the same lower bound, but different upper bounds (*Conjunto* reaches 80 while *Cardinal* is able to reach 78. By the way, *fd_sets* also only reaches 80.)

CLP(*FD*) presents better results than set solvers, but when using the union function of *Cardinal* (the most expressive version), although the upper bound was still larger than CLP (*FD*)'s (78 versus 76), the lower bound was significantly better (50 versus 9) and was easily reached thanks to the inferences on the cardinalities of the set elements.

Cardinal with the union function produced the smallest range among the CLP approaches. Furthermore, considering all CLP versions, we conclude that the optimum is in the range 50–76, with the lower bound coming from *Cardinal* with union, and the upper bound from CLP(*FD*). Naturally, CPLEX obtains a much better range: 54..63

We also compared the different set solvers for two other set covering problems from the same library: *scpe1* (50×500 matrix) and *scpclr*10 (511×210). While *Conjunto*, *fd_sets* and *Cardinal* behaved similarly, all three reaching the same bounds, *Cardinal* with the union function improved significantly the solutions, as shown in Table 13 with the reached ranges (again, we also include CPLEX results).

Ranges obtained using the union function are much smaller (especially in *scpe1*). Differences in execution times are also noteworthy: for instance, in *scpclr*10, value $n=8$ was proven impossible (thus obtaining 9 as best lower bound) in just one hundredth of a second (Pentium II/450) while simple *Cardinal* required more than 1 hour to prove it for $n=2$ (yet it was almost twice as fast as *Conjunto*). For the upper bound, the difference is not so large, a solution for $n=35$ being found in 16 seconds with the union function, and in 106 seconds without it.

With CPLEX, *scpe*1 was solved obtaining 5 as the minimum, and a much better range was obtained for *scpclr*10, namely 24..25, before execution aborted for running out of memory.

While recognising that set covering is an old and much studied problem with very efficient specialised ILP tools to solve it, we reckon that set reasoning is a more natural approach to deal

Table 13 Obtained ranges with or without union function

	Simple sets	With union fn.	CPLEX
scpe1	3..12	5..6	5
scpclr10	3..35	9..35	24..25

with it, and its application with set functions of *Cardinal* considerably improves other constraint approaches. We thus conjecture that a hybridisation of the two approaches with some interaction between solvers could add declarativity to efficiency and even improve it.

4.4 Steiner, Golfers, and Comparisons with Other Solvers

In this section we discuss modelling and solving of the well studied Steiner Triples and Social Golfers problems (numbered 44 and 10 in CSPlib) with set reasoning. These typical applications for set constraint solving also allow us to compare *Cardinal* with other recent set solvers, namely one that also considers set lexicographic bounds [30], and the one based on ROBDDs [24, 27]. We compare some results with those presented in these works, since such solvers are not publicly available.

4.4.1 Steiner Triples

The ternary Steiner problem of order n consists of finding a set of $n.(n-1)/6$ triples of distinct integer elements in $U=\{1,...,n\}$ such that any two triples have at most one common element. It is a hyper-graph problem coming from combinatorial mathematics where n modulo 6 has to be equal to 1 or 3 [13]. One possible solution for $n=7$ is {{1, 2, 3}, {1, 4, 5}, {1, 6, 7}, {2, 4, 6}, {2, 5, 7}, {3, 4, 7}, {3, 5, 6}}. The solution contains $7^*(7-1)/6=7$ triples. Steiner Triple Systems (STS) are a special case of Balanced Incomplete Block Designs (BIBD). In fact, a STS with n elements is a BIBD$(n, n.(n-1)/6, (n-1)/2, 3, 1)$ [13].

The problem can be formally formulated as finding sets S_i ($i \in \{1...t\}$) satisfying (61) and (62) below, where $t = n.(n-1)/6$ is the number of triples:

$$\forall_{i\in\{1...t\}}((S_i \subseteq U) \wedge \#S_i = 3). \tag{61}$$

$$\forall_{i,j\in\{1...t\},i\neq j}\#\left(S_i \cap S_j\right) \leq 1. \tag{62}$$

A CLP(*Sets*) approach directly models this problem by representing each triple as a set variable with upper bound U and cardinality 3, and constraining the cardinality of each intersection of a pair of triples not to be greater than 1. In contrast, an integer domain constraint satisfaction problem (CSP) model would use a variable for each set element with domain $\{1...n\}$, thus requiring the triple of variables, and far more constraints, in addition to not being so declarative. Furthermore, since then set elements are not automatically ordered, much symmetry that could occur in the integer approach is naturally eliminated with set variables.

Nevertheless, symmetry still occurs in the sets approach, since any two sets (triples) in a solution can be swapped without affecting it. For that reason, usually CLP(*Sets*) models add ordering constraints on the sets variables.

Since these solvers are not complete, a search phase must still occur to find a completely instantiated solution, and the way to do this may be critical on the computation time. Good heuristics for the order in which variables are picked for assignment and for the order in which values are assigned are then essential. For that, one should study the mathematical properties of the problem to find the best strategy. It is known that an element belongs to exactly $(n-1)/2$ triples [13]. We thus adopted the labelling strategy of deciding, for each element in turn, which were the $(n-1)/2$ sets it would be in and, consequently, the others it would be out of. This was simply done with the, respectively, Elem∈Set and Elem∉Set constraints of *Cardinal*.

Table 14 shows the results in seconds with this strategy (column *Cardinal*, on a Pentium 4, 2.4 GHz, 480 Mb RAM), compared with the ones presented in [30] and [27] for two

Table 14 Steiner triples results with set constraint solving. The '–' entries denote that no solution was obtained after 10 minutes

n	Hybrid	ROBDD	Cardinal
7	0.01	0.0	0.01
9	0.13	0.1	0.05
13	n.a.	80.2	0.61
15	1.06	2.4	0.91
19	n.a.	n.a.	7.94
21	n.a.	n.a.	39.07
25	n.a.	n.a.	–
27	n.a.	n.a.	–
31	99.63	n.a.	48.52
33	n.a.	n.a.	–

different set solvers with the same model, for *n* ranging from 7 to 33. *Hybrid* represents the set solver which, in addition to the usual set bounds, includes lexicographic bounds in an extended set domain (over ECLiPSe, Pentium 4, 2 GHz, 1 GB RAM); *ROBDD* represents the ROBDD-based set domain solver (Pentium-M 1.5 GHz), which allows merged constraints and no intermediate variables.

Hybrid added a redundant dual model as channelling constraints, which led to a labelling strategy similar to *Cardinal*'s, while *ROBDD* used the first-fail heuristic with an "element not in set" labelling (i.e. first try to exclude elements from a set domain). *ROBDD* presents no results for *n* higher than 15, and *Hybrid* only adds a result for n=31, skipping n=13, whereas *Cardinal* easily reaches 21 and also solves the n=31 instance.

Of course, labelling strategies are crucial in this sort of problems, according to the propagation used. In fact, a backtrack-free labelling for this problem is possible, as described on the internet (http://perso.wanadoo.fr/colin.barker/lpa/sts.htm, http://www.utu.fi/~honkala/cover.html). A labelling method is described based on the mathematical properties of the problem and the fact that Steiner triple systems can be constructed from a Latin square. One can then easily solve instances for *n* in the order of 1,000 and more. Therefore, the Steiner triples problem is not the best benchmark for CP due to the importance of labelling based on the mathematical study of the problem to be solved. In the next section we address another problem (social golfers), with published results for the ROBDD's based solver.

Notice that adding lexicographic bounds allow extra inferences that prune search space, reducing the number of backtracks-nevertheless, using the usual set bounds alone produce better runtime results, as also reported in [30]. There, the authors give the following example: from X::[{},{1, 2, 3, 4, 5}]$_3$, Y::[{},{1, 2, 3, 4}]$_{Cy:\{3,4\}}$, Z::[{},{1,2,3,4}]$_3$, $Z=X/\backslash Y$, the hybrid solver is able to infer that $5 \notin X$. Then they claim that a set solver without lexicographic bounds, such as *Cardinal*, could not make such inference. This is not so, since it would suffice to add a rule such as (63) to *Cardinal*:

$$\frac{Cx == Cz}{\{Z \subseteq X\} \mapsto \{X = Z\}} \tag{63}$$

Remember that set inclusion constraints are posted when the set intersection constraint is told.

4.4.2 Social Golfers

As taken from CSPlib: "The coordinator of a local golf club has come to you with the following problem. In her club, there are 32 social golfers, each of whom play golf once a week, and always in groups of 4. She would like you to come up with a schedule of play for these golfers, to last as many weeks as possible, such that no golfer plays in the same group as any other golfer on more than one occasion."

The problem generalizes to that of scheduling g groups of s golfers over w weeks, such that no golfer plays in the same group as any other golfer twice (i.e. maximum socialization is achieved). Thus, the golfers problem, given values w–g–s, can be formally defined as finding a set, *Weeks*, of sets, for the golfers in *Golfers*, such that:

$$\#\text{Weeks} = w \wedge \#\text{Golfers} = g \times s \wedge \forall_{\text{wk} \in \text{Weeks}} \bigcup_{\text{grp} \in \text{wk}} \text{grp} = \text{Golfers}. \tag{64}$$

$$\forall_{\text{wk} \in \text{Weeks}} \big(\#\text{wk} = g \wedge \forall_{g1,g2 \in \text{wk}, g1 \neq g2} (\#g_1 = s \wedge g_1 \cap g_2 = \emptyset)\big). \tag{65}$$

$$\forall_{w1,w2 \in \text{Weeks}, w1 \approx w2} \forall_{g1 \in w1} \forall_{g2 \in w2} \#(g_1 \cap g_2) \leq 1. \tag{66}$$

(Actually, the disjointness condition ($g1 \cap g2 = \emptyset$) in (65) is redundant in face of the others.)

A CLP(*Sets*) approach can model the w–g–s satisfaction problem with w*g set variables of cardinality s. Much symmetry occurs in this problem since any 2 week variables can be swapped, as well as any two groups in the same week, and even 2 values in the whole solution (e.g. replacing value 1 with 2, and vice-versa, using integers to represent golfers).

In this section we compare *Cardinal* with the results on this golfers problem presented in [27] for both *ROBDD*, as above, and *ic_sets* (another ECLiPSe sets solver, similar to *fd_sets*).

All three approaches order groups in each week with appropriate constraints. In particular, the *ROBDD* model uses a global constraint on the whole week, which partitions it from the set of golfers and simultaneously orders the groups. *Cardinal*, uses #< on the minimum of two sets, cooperating with the integer solver, thanks to the *Cardinal* facility to constrain set functions, and making use of the knowledge that sets to be ordered (groups)

Table 15 Runtime results of different set solvers for social golfers

w–g–s	Cardinal	ic_sets	ROBDD
2–5–4	0.83	5.3	0.1
2–6–4	1.75	35.5	0.2
2–7–4	2.82	70.3	0.6
2–8–5	–	–	3.1
3–5–4	1.89	9.3	0.5
3–6–4	4.62	59.2	2.3
3–7–4	6.37	113.6	3.5
4–5–4	3.13	10.5	1.3
4–6–5	–	–	171.5
4–7–4	12.46	135.8	21.8
4–9–4	42.45	22.7	338.4
5–4–3	**165.63**	–	**44.4**
5–5–4	28.65	267.3	4.4
5–7–4	17.18	–	54.7
5–8–3	1.01	4.1	6.6
6–4–3	**94.67**	–	**29.6**
6–5–3	–	–	2.0
6–6–3	1.20	2.7	2.5
7–5–3	–	–	28.4
7–5–5	–	–	**0.4**

have each s elements and are disjoint. Thus, in each week, for two groups, G_1 and G_2, G_1 comes before G_2 iff $\min(G_1)<\min(G_2)$.

Weeks can be ordered by comparisons on the first group. *Cardinal* did not adopt these ordering constraints since, in general, it led to poorer execution times.

Table 15 shows the results obtained with these three approaches for the instances of [27]. *Cardinal, ic_sets and ROBDD* run with the same machines and time limits as above. Rows in bold indicate instances with no possible solution. Times on such rows represent the time needed to prove it.

All three CLP(*Sets*) approaches use the "element in set" labelling. *Cardinal* and *ROBDD* obtained the best results with a first-fail heuristic, while *ic_sets* used sequential labelling.

Here, *ROBDD* obtains the best results due to its higher propagation having no intermediate variables, which greatly reduces search space. *ROBDD* is the only approach to solve all these instances, but, in general, results are dependent on "luck," according to each specific instance and strategy used. Even with the more consistent *ROBDD*, results fluctuate a bit, and are still far from corresponding to each problem model size. When applying sequential labelling, *ROBDD* does not obtain a solution in four instances. We also note that *Cardinal* results are consistently better than *ic_sets* (even considering a faster computer), solving three more instances.

We reckon however that such problems with sets of fixed cardinality are not the ones that benefit most from *Cardinal*'s inferences. Although less natural, these problems may also be modelled with integer variables, and solved with specialised tools. The mathematical properties of such highly structured problems should be further explored for the best results. In set constraint reasoning this may correspond to using powerful global constraints as has been used and proposed in the discussed solvers (*ROBDD* and *hybrid*).

5 Conclusions and Future Research

In this paper we presented a new general sets constraint solver just made available as an ECLiPSe Prolog library, *Cardinal*, which efficiently solves CSPs modelled with sets by performing a number of inferences over cardinality of sets. *Cardinal* improves significantly existing solvers based on sets, which allows efficient solving of digital circuits problems as well as general problems over sets. In addition, the extension of *Cardinal* to reason on different set functions other than cardinality (e.g. the union function) showed that other applications could benefit from a simple set modelling with such a powerful solver, thus allying declarativity with efficiency. In fact, set constraints are a very natural and concise way to express problems such as set covering, partitioning or bin-packing. Furthermore, actively considering set functions such as the cardinality, allows to efficiently solve these problems and to express many optimisation goals.

In particular, we addressed digital circuits' diagnostic related problems, which we showed how to model using signals combining Boolean and set values. We then presented a signal transformation to sets alone, which we proved to be correct, and demonstrated its usefulness with experimental results on differential diagnosis, where *Cardinal* outperformed other solvers by orders of magnitude. In the future, we intend to further explore such problems and investigate how similar techniques can efficiently be applied to different applications.

In this paper we have also seen that it was very easy to express set covering problems using an attached union function to a set (of sets) variable S. To express a set-partitioning problem instead, we need to ensure that the elements of S are pairwise disjoint. This can also be seen as a set function concerning each pair $<s_1, s_2>$ of elements of S (i.e. $disjoint(s_1, s_2)$). With such an extension, a set-partitioning problem could also be expressed with a single variable declaration, this time with three attached functions (cardinality, union and pairwise disjointness). Hence, in the future, *Cardinal* can be extended with set functions over all pairs or tuples of a variable, in addition to including other optional functions such as the minimum or maximum (already available in *Cardinal*) and more complex ones, and possibly accept user-defined functions and inferences.

Being able to express constraints over all pairs of a set variable, allows us to easily force a minimum distance between any two elements of a set of integers, for example. Of course, in this case, it is sufficient to force this distance between consecutive elements. Although a set is just a collection of unsorted elements, for efficiency reasons the system should provide facilities to consider consecutive elements and accept constraints between them (in fact, in practice *Cardinal* codes sets as sorted lists). With such facilities one can also easily force a maximum distance between two consecutive elements, which is extremely useful for scheduling applications. For instance, a timetable could be represented by a set variable where two consecutive courses should be close enough so that no big "holes" are generated. Then other constraints could be included such as the whole duration of the timetable (the difference between the maximum and the minimum function) being among some values, and so on.

Testing new set functions in *Cardinal* (or other set solver) on different types of problems thus seems to be an interesting research direction. Also, on the implementation level, a different and more efficient platform than ECLiPSe Prolog is currently under development (*CaSPER* [14]), where the integration of different solvers (e.g. SAT and FD) is being tested to fully benefit from each solver advantages, via channelling constraints in hybrid models. Improving or combining different domain representations (e.g. ROBDDs) may then also be more easily tested.

Acknowledgments Part of this work is fruit of my Ph.D. thesis and, for that, I thank my former supervisor Pedro Barahona. I also would like to thank the anonymous reviewers for their helpful comments, which contributed to improve this paper.

References

1. Aiken, A. (1994). Set constraints: results, applications and future directions. In Borning, A., ed., In *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, pages 326–335. Springer.
2. Azevedo, F. (2003). *Solving over multi-valued logics—application to digital circuits*. Frontiers of artificial intelligence and applications, ISBN: 1 58603 304 2, IOS, vol 91, xviii + 204 pages.
3. Azevedo, F., & Barahona, P. (1999). *Benchmarks for differential diagnosis*, at URL http://ssdi.di.fct.unl.pt/~fa/differential-diagnosis/benchmarks.html.
4. Azevedo, F., & Barahona, P. (2000). Applications of an extended set constraint solver. In *Proceedings of the 2000 ERCIM/CompulogNet Workshop on Constraints*.
5. Azevedo, F., & Barahona, P. (2000). Differentiating diagnostic theories through constraints over an eight-valued logic. In Horn, W., ed., *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI'2000)*, pages 73–77. IOS, Amsterdam.
6. Azevedo, F., & Barahona, P. (2000). Digital circuits problems with set constraints. In John Lloyd et al., ed., *Proceedings of the First International Conference on Computational Logic (CL'2000)*, pages 414–428. Springer.
7. Beasley, J. E. (1990). *OR-library: distributing test problems by electronic mail*, at URL http://mscmga.ms.ic.ac.uk/jeb/orlib, originally described in J. Oper. Res. Soc. 41(11):1069–1072.

8. Benhamou, F. (1995). *Interval constraint logic programming, in constraint programming: basics and trends*. In Podelski, A. ed., LNCS 910. Springer, March.
9. Birkhoff, G. (1967). *Lattice theory.* Colloquium Publications, American National Society, vol. 25.
10. Caseau, Y., Josset, F.-X., & Laburthe, F. (1999). CLAIRE: combining sets, search and rules to better express algorithms. *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 245–259.
11. Charatonik, W., & Podelski, A. (1996). The independence property of a class of set constraints. In Freuder, E. C., ed., *Proceedings of the 2nd International Conference on Principles and Practice of Constraint Programming (CP'96)*, pages 76–90. Springer.
12. Cheeseman, P., Kanefsky, B., & Taylor, W. (1991). Where the really hard problems are. *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, vol. 1, pages 331–337.
13. Colbourn, C. J., & Dinitz, J. H., ed., Steiner triple systems. *CRC Handbook of Combinatorial Designs.* CRC, pages 14−15 and 70, Boca Raton, FL.
14. Correia, M., Barahona, P., & Azevedo, F. (2005). CaSPER: A programming environment for development and integration of constraint solvers. In Azevedo et al. ed., *Proceedings of the First International Workshop on Constraint Programming Beyond Finite Integer Domains (BeyondFD'05)*, pages 59–73, 2005.
15. *CPLEX* (1988), in http://www.cplex.com.
16. Devienne, P., Talbot, J. M., & Tison, S. (1997). Solving classes of set constraints with tree automata. In Smolka, G., ed., *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 62–76. Springer.
17. ECRC (1994). *ECLiPSe (a) user manual, (b) extensions of the user manual.* Technical Report, ECRC.
18. Frühwirth, T. (1995). *Constraint handling rules, in constraint programming: basics and trends*. In Podelski, A., ed., LNCS 910. Springer.
19. Gent, I. P., & Walsh, T. (1999). *CSPLib: a benchmark library for constraints*. Technical report APES-09-1999. Available at http://www-users.cs.york.ac.uk/~tw/csplib. A shorter version appears in Proceedings of the 5th International Conference on Principles and Practices of Constraint Programming (CP-99).
20. Gervet, C. (1994). Conjunto: constraint logic programming with finite set domains. In Bruynooghe, M., ed., *International Symposium on Logic Programming*. Cambridge MA, MIT.
21. Gervet, C. (1997). *Interval propagation to reason about sets: definition and implementation of a practical language.* Constraints International Journal, vol. 1, no. 3, Kluwer, pages 191–244, March.
22. Gierz, G., & Hoffman, K. H., et al. (1980). *A compendium of continuous lattices*. Springer.
23. Graetzer, G. (1971). *LATTICE THEORY: first concepts and distributive lattices*. Freeman.
24. Hawkins, P., Lagoon, V., & Stuckey, P. J. (2004). Set bounds and (split) set domain propagation using ROBDDs. In *Proceedings of the 17th Australian Joint Conference on Artificial Intelligence, LNCS 3339*, pages 706−717. Springer, Berlin Heidelberg New York.
25. Heintze, N., & Jaffar, J. (1994). Set constraints and set-based analysis. In Borning, A., ed., *Proceedings of the 2nd International Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, pages 281–298. Springer.
26. ISCAS (1985). Special session on ATPG. In *Proceedings of the IEEE Symposium on Circuits and Systems*, pages 663–698, Kyoto, Japan, July.
27. Lagoon, V., & Stuckey, P. J. (2004). Set domain propagation using ROBDDs. *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming, LNCS 3258*, pages 347−361. Springer, Berlin Heidelberg New York.
28. Pacholski, L., & Podelski, A. (1997). Set constraints: a pearl in research on constraints. In Smolka, G., ed., *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP'97)*, pages 2–5, Springer.
29. Puget, J.-F. (1992). PECOS a high level constraint programming language. In *Proceedings of Spicis 92*.
30. Sadler, A., & Gervet, C. (2004). Hybrid set domains to strengthen constraint propagation and reduce symmetries. *Proceedings of Principles and Practice of Constraint Programming (CP 2004)*, vol. 3258.
31. Silva, L. G., Silveira, L. M., & Marques-Silva, J. P. (1999). Algorithms for solving boolean satisfiability in combinational circuits. *Proceedings of the IEEE/ACM Design and Test in Europe Conference (DATE)*.
32. Zhou, N.-F. (2000). Programming constraint propagation in reactive rules. *Proceedings of the 1st International Workshop on Rule-based Constraint Reasoning and Programming*.