Generating Propagators for Finite Set Constraints

Guido Tack[1], Christian Schulte[2], and Gert Smolka[1]

[1] PS Lab, Saarland University, Saarbrücken, Germany, {tack,smolka}@ps.uni-sb.de
[2] ECS, ICT, KTH - Royal Institute of Technology, Sweden, cschulte@kth.se

Abstract. Ideally, programming propagators as implementations of constraints should be an entirely declarative specification process for a large class of constraints: a high-level declarative specification is automatically translated into an efficient propagator. This paper introduces the use of existential monadic second-order logic as declarative specification language for finite set propagators. The approach taken in the paper is to automatically derive projection propagators (involving a single variable only) implementing constraints described by formulas. By this, the paper transfers the ideas of indexicals to finite set constraints while considerably increasing the level of abstraction available with indexicals. The paper proves soundness and completeness of the derived propagators and presents a run-time analysis, including techniques for efficiently executing projectors for n-ary constraints.

1 Introduction

Implementing constraints as propagators is an essential yet challenging task in developing and extending constraint programming systems. It is essential as the system's propagators define its efficiency, correctness, maintainability, and usability. It is challenging as most systems can only be programmed at a painfully low level of abstraction, requiring the use of languages such as C++, Java, or Prolog as well as intimate knowledge of complex programming interfaces. Some approaches that address this problem are indexicals (discussed below), high-level descriptions for the range and roots constraint [5], and deriving filtering algorithms from constraint checkers [3].

For finite domain constraints, indexicals have been introduced as a high-level programming language for projectors, a restricted form of propagators that only allow projections for a single variable [13, 6, 8]. While indexicals simplify programming propagators considerably, they have shortcomings with respect to level of abstraction and expressiveness. Indexicals still can not be automatically obtained from purely declarative specifications: multiple implementations for the same constraint are required, such as for different levels of consistency and entailment checking. Indexicals are only expressive enough for binary or ternary constraints. For n-ary constraints the very idea to decompose the propagator into projectors sacrifices efficiency. For example, decomposing a linear equation constraint involving n variables into n projectors increases the run-time from $O(n)$ to $O(n^2)$.

Worse still, for more involved constraint domains such as finite sets the burden on the programmer increases. Here programmers do not only need to reason on a by-value-base but on a by-set-base (typically involving sets for lower and upper bounds).

This additional complexity has been addressed by recent work of Ågren, Flener, and Pearson [2] in the context of constraint-based local search with finite set variables. The authors use formulas from an existential monadic second-order logic (∃MSO) to give descriptions of penalty functions to guide local search. While using ∃MSO as a high-level language, formulas are treated in a purely syntactic fashion: different penalty functions are obtained from syntactically different yet logically equivalent formulas.

Main contribution. This paper presents an exercise in applied software engineering. We provide a high-level, abstract specification language for finite set constraints, and an automatic translation into an efficient implementation.

The paper introduces the use of ∃MSO as a purely declarative specification language for automatically obtaining propagators for finite set constraints. As an example, consider the binary union constraint $x = y \cup z$. In the fragment of ∃MSO we define, this constraint can be written $\forall v.v \in x \leftrightarrow v \in y \vee v \in z$, where v ranges over individual values from the universe, and x, y, and z are the finite set variables. Formulas allow us to specify constraints in a *purely declarative* way.

From a constraint expressed as a formula, we derive a set of *projectors*, one for each of the variables. For the binary union example, we get $p_x = (y \cup z \subseteq x \subseteq y \cup z)$, $p_y = (x \setminus z \subseteq y \subseteq x)$, $p_z = (x \setminus y \subseteq z \subseteq x)$. Projectors transfer the idea of finite domain indexicals to finite sets. We show that the generated set of projectors is sound and complete for the constraint it was derived from.

Projectors are an *executable specification*. Just like indexicals, projectors can be implemented efficiently. We have implemented an interpreter and compiler for projectors for the Gecode library [20]. We can specify constraints purely declaratively as formulas, and then generate an implementation that is provably sound and complete.

Plan of the paper. After presenting a framework for studying propagators in Section 2, this paper is organized as follows:

- The use of ∃MSO as a declarative specification language for finite set constraints is introduced (Section 3).
- Projectors for finite set constraints are derived from range expressions over finite set variables (Section 4). This transfers the ideas of indexicals to finite set constraints.
- A correspondence between formulas in ∃MSO and finite set projectors is established (Section 5), where equivalent formulas lead to equivalent projectors. We prove that the derived projectors are correct and as strong as possible.
- Negated and reified constraints are obtained from entailment-checking projectors (Section 6).
- The specification language is shown to be also suitable for generating ROBDD-based propagators (Section 7).
- Techniques for evaluating projectors derived from range expressions are discussed, including common subexpression elimination (Section 8).
- We analyze the run-time of projectors and generalize range expressions to propagators involving arbitrary numbers of set variables (Section 9).

Section 10 concludes the paper.

2 Constraints and Propagators

This section introduces the central notions used in the rest of the paper. The presentation builds on the extended constraint systems of Benhamou [4].

Variables and constraints. We assume a finite set of variables *Var* and a finite set of values *Val*. Constraints are characterized by assignments $\alpha \in Asn$ mapping variables to values: $Asn = Var \to Val$. A constraint $c \in Con$ is a set of fulfilling assignments, $Con = \mathscr{P}(Asn)$. Basing constraints on assignments (defined for all variables *Var*) rather than tuples or partial assignments (defined for a subset of variables $X \subseteq Var$) simplifies further presentation. Note that a set of tuples or partial assignments for a set of variables X can be extended easily to a set of assignments by mapping all variables from $Var \setminus X$ to all possible values.

Domains and stores. Propagation is performed by propagators over constraint stores (or just stores) where stores are constructed from domain approximations as follows. A domain $d \in Dom$ contains values a variable can take, $Dom = \mathscr{P}(Val)$. A domain approximation A for *Dom* is a subset of *Dom* that is closed under intersection and that contains at least those domains needed to perform constraint propagation, namely $\emptyset$, *Val*, and all singletons $\{v\}$ (called *unit approximate domains* in [4]). We call elements of A approximate domains.

A store $S \in Store$ is a tuple of approximate domains. The set of stores is a Cartesian product $Store = A^n$, where $n = |Var|$ and A is a domain approximation for *Dom*.

Note that a store S can be identified with a mapping $S \in Var \to A$ or with a set of assignments $S \in \mathscr{P}(Asn)$. This allows us to treat stores as constraints when there is no risk of confusion. In particular, for any assignment α, $\{\alpha\}$ is a store.

A store S_1 is *stronger* than a store S_2, if $S_1 \subseteq S_2$. By $(c)_{Store}$ we refer to the strongest store including all values of a constraint, defined as $\min\{S \in Store | c \subseteq S\}$. The minimum exists as stores are closed under intersection. Note that this is a meaningful definition as stores only allow Cartesian products of approximate domains. Now, for a constraint c and a store S, $(c \cap S)_{Store}$ refers to removing all values from S not supported by the constraint c (with respect to the approximative nature of stores). For a constraint c, we define $c.x = \{v | \exists \alpha \in c : \alpha(x) = v\}$ as the projection on the variable x. For a store S, $S.x$ is the x-component of the tuple S.

Constraint satisfaction problems. A constraint satisfaction problem is a pair $(C,S) \in \mathscr{P}(Con) \times Store$ of a set of constraints C and a store S. A solution of a constraint satisfaction problem (C,S) is an assignment α such that $\alpha \in S$ and $\alpha \in \bigcap_{c \in C} c$.

Propagators. Propagators serve here as implementations of constraints. They are sometimes also referred to as constraint narrowing operators or filter functions. A propagator is a function $p \in Store \to Store$ that is contracting ($p(S) \subseteq S$) and monotone ($S' \subseteq S \Rightarrow p(S') \subseteq p(S)$). Note that propagators are not required to be idempotent.

A propagator is sound for a constraint c iff for all assignments α, we have $c \cap \{\alpha\} = p(\{\alpha\})$. This implies that for any store S, we have $(c \cap S)_{Store} \subseteq p(S)$. Thus, p is sound for c if it does not remove solutions and can decide if an assignment fulfills c or not.

A propagator is complete for a constraint c iff for all stores S, we have $(c \cap S)_{Store} = p(S)$. A complete propagator thus removes all assignments from S that are locally inconsistent with c. Note that a complete propagator for a constraint c establishes domain-consistency for c with respect to the domain approximation.

Fixpoints as propagation results. With set inclusion lifted point-wise to Cartesian products, stores form a complete partial order. Hence, one can show that the greatest mutual fixpoint of a set of propagators P for a store S exists. We write the fixpoint as $\bigsqcap_P S$.

In the following we will be interested in the soundness and completeness of sets of propagators. A set of propagators P is sound for a constraint c iff $\forall \alpha : c \cap \{\alpha\} = \bigsqcap_P \{\alpha\}$. Likewise, P is complete for c iff $\forall S : (c \cap S)_{Store} = \bigsqcap_P S$.

Note that we specify *what* is computed by constraint propagation and not *how*. Approaches how to perform constraint propagation can be found in [4, 1, 18].

Projectors. Projection propagators (or projectors, for short) behave as the identity function for all but one variable. Thus, for a projector on x, we have $\forall S\ \forall y \neq x : (p(S)).y = S.y$. To simplify presentation, we write a projector on x as $p_x \in Store \to A$.

3 A Specification Language for Finite Set Constraints

This section introduces a high-level, declarative language that allows to specify finite set constraints intensionally. Finite set constraints talk about variables ranging over finite sets of a fixed, finite universe: $Val = \mathscr{P}(\mathscr{U})$. To specify finite set constraints, we use a fragment of existential monadic second order logic (∃MSO).

In the following sections, we use ∃MSO for proving properties of projectors and the constraints they implement. Furthermore, we derive sound and complete projectors from formulas.

3.1 A logic for finite set constraints

In our framework, constraints are represented extensionally as collections of assignments. To be able to reason about them, we use formulas of a fragment of second-order logic as an intensional representation.

Finite set constraints can be elegantly stated in existential monadic second-order logic (∃MSO), see [2] for a discussion in the context of local search. Second-order variables take the role of finite set variables from *Var*, and first-order variables range over individual values from the universe $\mathscr{U}$.

We use the fragment defined by the grammar in Table 1. It has the usual Boolean connectives, a first-order universal quantifier and a second-order existential quantifier, all with their standard interpretation. As the second-order variables represent finite sets, we write $v \in x$ instead of $x(v)$ and abbreviate $\neg v \in x$ as $v \notin x$. Furthermore, we use implication ($\to$) and equivalence ($\leftrightarrow$) as the usual abbreviations.

S	$::= \exists x\ S \mid F$	*second-order quantification*
F	$::= \forall v.B \mid F \wedge F$	*first-order quantification*
B	$::= B \wedge B \mid B \vee B \mid \neg B \mid v \in x \mid \bot$	*basic formulas*

Table 1. Syntax of a fragment of ∃MSO

Formulas as constraints. The extension of a formula φ is the set of models of φ. In the case of monadic second-order formulas without free first-order variables, a model corresponds directly to an assignment. The extension of φ is hence the set of assignments that satisfy φ.

A set of satisfying (or fulfilling) assignments is exactly the definition of a constraint. Thus, for a formula φ without free first-order variables, we write $[\varphi]$ for its extension, and we use it like a constraint. Table 2 shows some examples of finite set constraints expressed as ∃MSO formulas.

Some important constraints, like disequality of sets, cannot be expressed using this logic, as they require a first-order existential quantifier. Extending the logic with existential quantification is discussed in Section 6.

$x \subseteq y$	$[\forall v.v \in x \to v \in y]$	*subset*
$x = y$	$[\forall v.v \in x \leftrightarrow v \in y]$	*equality*
$x = y \cup z$	$[\forall v.v \in x \leftrightarrow v \in y \vee v \in z]$	*union*
$x = y \cap z$	$[\forall v.v \in x \leftrightarrow v \in y \wedge v \in z]$	*intersection*
$x \parallel y$	$[\forall v.v \notin x \vee v \notin y]$	*disjointness*

Table 2. Finite set constraints expressed in ∃MSO

4 Finite Integer Set Projectors

This section introduces a high-level programming language for implementing projectors for finite set constraints. First, we define domain approximations for finite set constraints – this yields a concrete instance of the framework defined in Section 2. Then we describe range expressions as introduced by Müller [15], which carry over the ideas behind finite domain indexicals to finite set projectors.

4.1 Domain approximations for sets

With $Val = \mathscr{P}(\mathscr{U})$ for set variables, a full representation of the domain of a variable can be exponential in size. This makes domain approximations especially important for set variables.

$$
\begin{array}{lll}
r_{\mathrm{glb}}(x,S) & = \mathrm{glb}(S.x) & r_{\mathrm{glb}}\left(\overline{R},S\right) = \overline{r_{\mathrm{lub}}(R,S)} \\
r_{\mathrm{glb}}(R_1 \cup R_2,S) & = r_{\mathrm{glb}}(R_1,S) \cup r_{\mathrm{glb}}(R_2,S) & r_{\mathrm{glb}}(\emptyset,S) = r_{\mathrm{lub}}(\emptyset,S) = \emptyset \\
r_{\mathrm{glb}}(R_1 \cap R_2,S) & = r_{\mathrm{glb}}(R_1,S) \cap r_{\mathrm{glb}}(R_2,S) & \\
 & & \\
r_{\mathrm{lub}}(R,S) & \textit{analogous} &
\end{array}
$$

Table 3. Evaluating range expressions in a store

Most constraint programming systems use *convex sets* as an approximation (introduced in [16], formalized in [10]). A convex set d is a set of sets that can be described by a greatest lower bound $\mathrm{glb}(d)$ and a least upper bound $\mathrm{lub}(d)$ and represents the sets $\{v \mid \mathrm{glb}(d) \subseteq v \subseteq \mathrm{lub}(d)\}$.

In our terminology, we define the domain approximation A_{Set} as the set of all convex sets. A_{Set} is indeed a domain approximation, as $\emptyset$, *Val*, and all singletons $\{s\}$ for $s \in \mathit{Val}$ are convex, and the intersection of two convex sets is again convex. We write $\mathrm{glb}(S.x)$ and $\mathrm{lub}(S.x)$ to denote the greatest lower resp. least upper bound of x in the store S.

4.2 Range expressions for finite set projectors

Given A_{Set}, a projector for a finite set variable x can be written as a function $p_x \in \mathit{Store} \to \mathit{Val} \times \mathit{Val}$, returning the pruned greatest lower and least upper bound for x.

For finite domain constraints, indexicals have proven a useful projector programming language. The main idea goes back to cc(FD) [12, 13] and was later elaborated in the context of clp(FD), AKL, and SICStus Prolog [9, 6, 8]. Indexicals build on *range expressions* as a syntax for set-valued expressions that can be used to define the projection of a constraint. These ideas easily carry over to finite set projectors over A_{Set}. We define range expressions by the following grammar:

$$R \quad ::= x \mid R \cup R \mid R \cap R \mid \overline{R} \mid \emptyset$$

A finite set projector for the variable x can now be defined by two range expressions, one pruning the greatest lower bound of x, and one pruning the upper bound. We write such a projector $p_x = (R_1 \subseteq x \subseteq R_2)$.

Range expressions are evaluated in a store using the functions r_{glb} and r_{lub} from Table 3. A projector $p_x = (R_1 \subseteq x \subseteq R_2)$ is defined to compute the function $p_x(S) = (r_{\mathrm{glb}}(R_1,S) \cup \mathrm{glb}(S.x), r_{\mathrm{lub}}(R_2,S) \cap \mathrm{lub}(S.x))$.

Proposition 1. *A function $p_x = (R_1 \subseteq x \subseteq R_2)$ is contracting and monotone and therefore a propagator.*

Proof. The function p_x is contracting by construction. It is monotone iff $\forall S' \subseteq S : p_x(S') \subseteq p_x(S)$, or equivalently $\forall S' \subseteq S : \mathrm{glb}(p_x(S)) \subseteq \mathrm{glb}(p_x(S'))$ and $\mathrm{lub}(p_x(S')) \subseteq \mathrm{lub}(p_x(S))$. For a projector defined by range expressions $(R_1 \subseteq x \subseteq R_2)$, we must have $\forall S' \subseteq S : r_{\mathrm{glb}}(R_1,S) \subseteq r_{\mathrm{glb}}(R_1,S')$ and $r_{\mathrm{lub}}(R_2,S') \subseteq r_{\mathrm{lub}}(R_2,S)$. This can be shown by induction over the structure of range expressions.

$$
\begin{array}{lll}
e_v(x) & = v \in x & e_v\left(\overline{R}\right) = \neg e_v(R) \\
e_v(R_1 \cup R_2) & = e_v(R_1) \vee e_v(R_2) & e_v(\emptyset) \;\; = \bot \\
e_v(R_1 \cap R_2) & = e_v(R_1) \wedge e_v(R_2) &
\end{array}
$$

Table 4. Translating a range expression to a formula

5 ∃MSO Specifications for Projectors

We have seen two specification languages so far, one high-level declarative language for specifying set constraints, and one programming language for set projectors. This section shows how to connect the two languages: on the one hand, we want to find a formula describing the constraint a projector implements, on the other hand, we want to find projectors implementing the constraint represented by a formula.

We derive a ∃MSO formula φ for a given projector p_x such that p_x is sound for $[\varphi]$. The formula thus declaratively describes the constraint that p_x implements.

For the other direction, we generate a set of projectors P for a given formula φ such that P is sound and complete for $[\varphi]$. This allows us to use ∃MSO as a specification language for sound and complete finite set projectors.

5.1 From range expressions to ∃MSO specifications

Given a projector p_x, we now derive a formula φ_{p_x} such that p_x is sound for $[\varphi_{p_x}]$.

The correspondence between relational algebra and logic gives rise to the definition of the function e (Table 4). For a range expression R, $e_v(R)$ is a formula that is true iff $v \in R$. Furthermore, a subset relation corresponds to a logical implication. Hence, for a projector $p_x = (R_1 \subseteq x \subseteq R_2)$, we define $\varphi_{p_x} = \forall v.\, (e_v(R_1) \rightarrow v \in x) \wedge (v \in x \rightarrow e_v(R_2))$. We can now show that p_x is sound for $[\varphi_{p_x}]$.

Lemma 1. *Range expressions have the same extension as the corresponding formulas.*

$$
\begin{array}{l}
\alpha \in [\forall v. e_v(R) \rightarrow v \in x] \Leftrightarrow r_{\text{glb}}(R, \{\alpha\}) \subseteq \alpha(x) \\
\alpha \in [\forall v. v \in x \rightarrow e_v(R)] \Leftrightarrow \alpha(x) \subseteq r_{\text{lub}}(R, \{\alpha\})
\end{array}
$$

Proof. By induction over the structure of range expressions.

Proposition 2. *Every projector $p_x = (R_1 \subseteq x \subseteq R_2)$ is sound for the constraint $[\varphi_{p_x}]$.*

Proof. We have to show $[\varphi_{p_x}] \cap \{\alpha\} = p_x(\{\alpha\})$ for all α. This is equivalent to showing $\alpha \in [\varphi_{p_x}] \Leftrightarrow p_x(\{\alpha\}) = \{\alpha\}$. From the definition of projectors, we get $p_x(\{\alpha\}) = \{\alpha\} \Leftrightarrow r_{\text{glb}}(R_1, \{\alpha\}) \subseteq \alpha(x) \wedge \alpha(x) \subseteq r_{\text{lub}}(R_2, \{\alpha\})$. Lemma 1 says that this is equivalent to $\alpha \in [\forall v. e_v(R_1) \rightarrow v \in x]$ and $\alpha \in [\forall v. v \in x \rightarrow e_v(R_2)]$. This can be combined into $\alpha \in [\forall v.\, (e_v(R_1) \rightarrow v \in x) \wedge (v \in x \rightarrow e_v(R_2))] = [\varphi_{p_x}]$.

Equivalence of projectors. We say that two projectors p_x and p'_x are equivalent iff they are sound for the same constraint. Using the translation to formulas as introduced above, p_x and p'_x are equivalent iff $\varphi_{p_x} \equiv \varphi_{p'_x}$. Note that two equivalent projectors may still differ in propagation strength, for instance if only one of them is complete for φ_{p_x}.

5.2 From specifications to projectors

The previous section shows that for every set projector p_x one can find a formula φ_{p_x} such that p_x is sound for $[\varphi_{p_x}]$. We now want to find a set of projectors P for a given formula φ such that P is sound and complete for $[\varphi]$. Remember that we need a set of projectors, as each individual projector only affects one variable. For completeness, all variable domains have to be pruned.

A first step extracts all implications to a single variable x from a given formula φ, yielding a normal form for φ. A second step then transforms this normal form into a projector for x. Using the transformation to normal form, we can construct a set of projectors, one for each variable in φ. Finally, we show that the set of projectors obtained this way is complete for $[\varphi]$.

Step 1: Extraction of implications. As we use the convex-set approximation, a projector for a variable x has to answer two questions: which values for x occur in all assignments satisfying the constraint (the greatest lower bound), and which values can occur at all in a satisfying assignment (the least upper bound). The idea of our transformation is that the first question can be answered by a formula of the form $\forall v.\psi_1 \rightarrow v \in x$, while the second question corresponds to a formula $\forall v.v \in x \rightarrow \psi_2$. The remaining problem is hence to transform any formula φ into an equivalent $\varphi' = \bigwedge_x \forall v : (\psi_{1_x} \rightarrow v \in x) \wedge (v \in x \rightarrow \psi_{2_x})$.

The transformation proceeds in four steps: (1) Skolemization, (2) merging of first-order quantifiers, (3) transformation to conjunctive normal form, and (4) extraction of implications for each variable.

Skolemization removes all second-order existential quantifiers and replaces them with fresh variables. Intuitively, the fresh variables play the role of intermediate variables. Though this is not strictly an equivalence transformation, the extension of the resulting formula is the same for the variables of the original formula. We can *merge universal quantifiers* $(\forall v.\psi) \wedge (\forall v.\psi')$ into $\forall v.\psi \wedge \psi'$. After these transformations, we arrive at a formula of the form $\forall v.\psi$ where ψ is quantifier-free. We can then transform ψ into conjunctive normal form (CNF), into a set of *clauses* $\{C_1,\ldots,C_n\}$, where each C_i is a set of *literals* L (either $v \in x$ or $v \notin x$).

From the CNF, one can extract all implications for a variable x as follows:

$$\begin{aligned}
\forall v.\psi &\equiv \forall v. \textstyle\bigwedge_i \bigvee_{L' \in C_i} L' \\
&\equiv \forall v. \textstyle\bigwedge_i \left(\bigwedge_{L' \neq (v \in x) \in C_i} \overline{L'} \rightarrow v \in x \right) \wedge \left(\bigwedge_{L' \neq (v \notin x) \in C_i} \overline{L'} \rightarrow v \notin x \right) \\
&\equiv \forall v. \left(\bigvee_i \bigwedge_{L' \neq (v \in x) \in C_i} \overline{L'} \rightarrow v \in x \right) \wedge \left(v \in x \rightarrow \bigwedge_i \bigvee_{L' \neq (v \notin x) \in C_i} L' \right)
\end{aligned}$$

If ψ does not contain $v \in x$ (or $v \notin x$), the corresponding implication is trivially true. Thus, in practice, this transformation only has to consider the free variables of ψ (after Skolemization) instead of all the variables.

We call this form L-implying normal form of φ, written $INF_L(\varphi)$. We refer to $\bigvee_i(\bigwedge_{L' \neq (v \in x) \in C_i} \overline{L'})$ as ψ_{1_x} and to $\bigwedge_i \bigvee_{L' \neq (v \notin x) \in C_i} L'$ as ψ_{2_x}.

Step 2: Compilation to projectors. The two subformulas of an L-implying normal form, ψ_{1_x} and ψ_{2_x}, are quantifier-free and contain a single first-order variable v. We observe that the function e_v from Table 4 has an inverse e^{-1} for quantifier-free formulas with a single free first-order variable v.

With Proposition 2 we can thus argue that $(e^{-1}(\psi_{1_x}) \subseteq x \subseteq e^{-1}(\psi_{2_x}))$ is sound for $[INF_x(\varphi)]$. Furthermore, for any formula $\varphi' = \bigwedge_x INF_x(\varphi)$, it is easy to show that the set $P = \{(e^{-1}(\psi_{1_x}) \subseteq x \subseteq e^{-1}(\psi_{2_x})) \mid x \in Var\}$ is sound for $[\varphi]$.

Example. Consider the ternary intersection constraint $x = y \cap z$. It can be expressed in $\exists$MSO as $\forall v.v \in x \leftrightarrow v \in y \wedge v \in z$. The implied normal forms for x, y, and z are

$\forall v.\ (v \in y \wedge v \in z \rightarrow v \in x)\ \wedge\ (v \in x \rightarrow v \in y \wedge v \in z)$
$\forall v.\ (v \in x \rightarrow v \in y)\ \wedge\ (v \in y \rightarrow v \in x \vee v \notin z)$
$\forall v.\ (v \in x \rightarrow v \in z)\ \wedge\ (v \in z \rightarrow v \in x \vee v \notin y)$

Deriving projectors from these formulas, we get $p_x = (y \cap z \subseteq x \subseteq y \cap z)$, $p_y = (x \subseteq y \subseteq x \cup \bar{z})$, and $p_z = (x \subseteq z \subseteq x \cup \bar{y})$.

Proposition 3. *We can now prove completeness of the generated projector set: Given a formula $\varphi' = \bigwedge_x INF_x(\varphi)$, the projector set $P = \{(e^{-1}(\psi_{1_x}) \subseteq x \subseteq e^{-1}(\psi_{2_x})) \mid x \in Var\}$ is complete for $[\varphi]$.*

Proof. We only sketch the proof due to space limitations. For completeness, we have to show $\bigsqcap_P S \subseteq ([\varphi] \cap S)_{Store}$. If $([\varphi] \cap S)_{Store} = S$, this is trivial. Otherwise, we only have to show that at least one projector can make a contribution.

First, we can show that if $[\varphi]$ removes values from the greatest lower or least upper bounds of variables in S, then there exists at least one x such that already $[INF_x(\varphi)]$ prunes S. More formally, $([\varphi] \cap S)_{Store} \neq S$ implies $\mathrm{glb}(([\forall v.\psi_{1_x} \rightarrow v \in x] \cap S).x) \neq \mathrm{glb}(S.x)$ or $\mathrm{lub}(([\forall v.v \in x \rightarrow \psi_{2_x}] \cap S).x) \neq \mathrm{lub}(S.x)$ for some x. This is true because of the way implications are extracted from φ.

The second step in the proof is to show that if $\mathrm{glb}(([\forall v.\psi_{1_x} \rightarrow v \in x] \cap S).x) \neq \mathrm{glb}(S.x)$, then $\mathrm{glb}(([\forall v.\psi_{1_x} \rightarrow v \in x] \cap S).x) \subseteq r_{\mathrm{glb}}(e^{-1}(\psi_{1_x}), S) \cup \mathrm{glb}(S.x)$, and dually for lub. This means that the projector for x makes a contribution and narrows the domain.

Finally, if none of the projectors can contribute for the store S, we know on the one hand that $([\varphi] \cap S)_{Store} = S$ (we have just proved that), and on the other hand that $\bigsqcap_P S = S$ (S must be a fixpoint). This concludes the proof.

With Propositions 2 and 3, it follows that the set of projectors $\{p_x = (e^{-1}(\psi_{1_x}) \subseteq x \subseteq e^{-1}(\psi_{2_x})) \mid x \in Var\}$ is sound and complete for $[\bigwedge_x INF_x(\varphi)] = [\varphi]$.

Finding small formulas. The projectors derived from two equivalent formulas $\varphi \equiv \varphi'$ are equivalent in the sense that they are sound and complete for both formulas. However, the size of their range expressions, and therefore the time complexity of propagating them (as discussed in Section 9), can differ dramatically.

The antecedents in an L-implied normal form are in disjunctive normal form (DNF). We can apply well-known techniques from circuit minimization to find equivalent minimal formulas (e.g. the Quine-McCluskey and Petrick's methods). Note that for the "standard constraints" like those from Table 2, the generated *INF* are minimal.

6 Negated and Reified Constraints

In this section, we show how to specify and execute negated and reified constraints, by transferring the ideas of entailment checking indexicals [7] to finite set projectors. Negation adds first-order existential quantifiers to the specification language.

Checking entailment of a projector. A propagator p is called *entailed* by a store S iff for all stores $S' \subseteq S$ we have $p(S') = S'$.

In the indexical scheme, entailment can be checked using anti-monotone indexicals [7]. Following this approach, we use the anti-monotone interpretation of a projector to check its entailment. For example, a sufficient condition for $p = (R_1 \subseteq x \subseteq R_2)$ being entailed is $r_{\text{lub}}(R_1, S) \subseteq \text{glb}(S.x)$ and $\text{lub}(S.x) \subseteq r_{\text{glb}}(R_2, S)$.

Negated projectors – existential quantifiers. A negated projector $\overline{p}$ can be propagated by checking entailment of p. If p is entailed, $\overline{p}$ is failed, and vice versa. Such a $\overline{p}$ is sound for $[\neg\varphi_p]$, but not necessarily complete.

We observe that this gives us a sound implementation for formulas $\neg\varphi$, where $\varphi = \forall v.\,(\psi_1 \rightarrow v \in x) \wedge (v \in x \rightarrow \psi_2)$. This is equivalent to $\neg\varphi = \exists v.\neg(\psi_1 \rightarrow v \in x \wedge v \in x \rightarrow \psi_2)$. We can thus extend our formulas with existential first-order quantification:

$$F \quad ::= \forall v.B \mid \boxed{\exists v.B} \mid F \wedge F$$

One important constraint we could not express in Section 3 was disequality of sets, $x \neq y$. Using existential quantification, this is easy: $\exists v.\neg(v \in x \leftrightarrow v \in y)$.

Reified constraints. A reified constraint is a constraint that can be expressed as a formula $\varphi \leftrightarrow b$, for a $0/1$ finite domain variable b. Exactly as for reified finite domain constraints implemented as indexicals [8], we can detect entailment and dis-entailment of φ, and we can propagate φ and $\neg\varphi$. Thus, we can reify any constraint expressible in our $\exists$MSO fragment.

7 Generating Projectors for BDD-based Solvers

Solvers based on binary decision diagrams (BDDs) have been proposed as a way to implement full domain consistency for finite set constraints [11]. This section briefly recapitulates how BDD-based solvers work. We can then show that $\exists$MSO can also be used as a specification language for BDD-based propagators.

Domains and constraints as Boolean functions. The solvers as described by Hawkins et al. [11] represent both the variable domains and the propagators as Boolean functions.

A finite integer set s can be represented by its characteristic function: $\chi_s(i) = 1 \Leftrightarrow i \in s$. A domain, i.e. a set of sets, is a disjunction of characteristic functions.

Constraints can also be seen as Boolean functions. In fact, formulas in our $\exists$MSO fragment are a compact representation of Boolean functions. The universal quantification $\forall v$ corresponds to a finite conjunction over all v, because the set of values *Val* is finite. For instance, the formula $\forall v.v \in x \rightarrow v \in y$, modeling the constraint $x \subseteq y$, can be transformed into the Boolean function $\bigwedge_v x_v \rightarrow y_v$.

Reduced Ordered Binary Decision Diagrams. ROBDDs are a well-known data structure for storing and manipulating Boolean functions. Hawkins et al. propose to store complete variable domains and propagators as ROBDDs. Although this representation may still be exponential in size, it works well for many practical examples.

Propagation using ROBDDs also performs a projection of a constraint on a single variable, with respect to a store. The fundamental difference to our setup is the choice of domain approximation, as ROBDDs allow to use the full $A = \mathscr{P}(Dom)$.

Hawkins et al. also discuss approximations including cardinality information and lexicographic bounds [17]. These approximations can yield stronger propagation than simple convex bounds but, in contrast to the full domain representation, have guaranteed polynomial size.

From specification to ROBDD. As we have sketched above, ∃MSO formulas closely correspond to Boolean functions, and can thus be used as a uniform specification language for projectors based on both range expressions and BDDs.

Although BDDs can be used to implement the approximation based on convex sets and cardinality, our approach still has some advantages: (1) It can be used for existing systems that do not have a BDD-based finite set solver. (2) A direct implementation of the convex-set approximation may be more memory efficient. (3) Projectors can be compiled statically, and independent of the size of $\mathscr{U}$. Still, projectors based on range expressions offer the same compositionality as BDD-based projectors.

8 Implementing Projectors

The implementation techniques developed for finite domain indexicals directly carry over to set projectors. We thus sketch only briefly how to implement projectors. Furthermore, we present three ideas for efficient projector execution: grouping projectors, common subexpression elimination, and computing without intermediate results.

Evaluating projectors. The operational model of set projectors is very similar to that of finite domain indexicals. We just have to compute two sets (lower and upper bound) instead of one (the new domain).

The main functionality a projector $(R_1 \subseteq x \subseteq R_2)$ has to implement is the evaluation of $r_{\text{glb}}(R_1, S)$ and $r_{\text{lub}}(R_2, S)$. Just like indexicals, we can implement r_{glb} and r_{lub} using a stack-based interpreter performing a bottom-up evaluation of a range expression [6], or generate code that directly evaluates ranges [9, 15].

Grouping projectors. Traditionally, one projector (or indexical) is treated as one propagator. Systems like SICStus Prolog [14] schedule indexicals with a higher priority than propagators for global constraints.

However, from research on virtual machines it is well known that grouping several instructions into one *super-instruction* reduces the dispatch overhead and possibly enables an optimized implementation of the super-instructions.

In constraint programming systems, propagators play the role of instructions in a virtual machine. Müller [15] proposes a scheme where the set of projectors that implements one constraint is compiled statically into one propagator.

Common Subexpression Elimination. Range expressions form a tree. A well-known optimization for evaluating tree-shaped expressions is *common subexpression elimination* (or CSE): if a sub-expression appears more than once, only evaluate it once and reuse the result afterwards.

Using CSE for range expressions has been suggested already in earlier work on indexicals (e.g. [6]). Common subexpressions can be shared both within a single range expression, between the two range expressions of a set projector, and between range expressions of different projectors.

If subexpressions are shared between projectors, special care must be taken of the order in which the projectors are evaluated, as otherwise evaluating one projector may invalidate already computed subexpressions. This makes grouping projectors a prerequisite for inter-projector CSE, because grouping fixes the order of evaluation.

Using iterators to evaluate range expressions. Bottom-up evaluation of range expressions usually places intermediate results on a stack. This imposes a significant performance penalty for finite set constraints, which can be avoided using *range iterators* [19]. Range iterators allow to evaluate range expressions without intermediate results.

A range iterator represents a set of integers implicitly by providing an interface for iterating over the maximal disjoint ranges of integers that define the set. Set variables can provide iterators for their greatest lower and least upper bound. The union or intersection of two iterators can again be implemented as an iterator. Domain operations on variables, such as updating the bounds, can take iterators as their arguments. Iterators can serve as the basis for both compiling the evaluation of range expressions to e.g. C++, and for evaluating range expressions using an interpreter.

Implementation in Gecode. We have implemented an interpreter and a compiler for finite set projectors for the Gecode library ([20], version 1.3 or higher). Both interpreter and compiler provide support for negated and reified propagators.

The interpreter is a generic Gecode propagator that can be instantiated with a set of projectors, defined by range expressions. Propagation uses range iterators to evaluate the range expressions. This approach allows to define new set propagators at run-time.

The compiler generates C++ code that implements a Gecode propagator for a set of projectors. Again, range iterators are used for the actual evaluation of range expressions.

Compiling to C++ code has three main advantages. First, the generated code has no interpretation overhead. Second, more expensive static analysis can be done to determine a good order of propagation for the grouped projectors (see [15]). Third, the generated code uses template-based polymorphism instead of virtual function calls (as discussed in [19]). This allows the C++ compiler to perform aggressive optimizations.

9 Run-time Analysis

In this section, we analyze the time complexity of evaluating set projectors. This analysis shows that the naive decomposition of a constraint over n variables into n projectors leads to quadratic run-time $O(n^2)$. We develop an extended range expression language that allows to evaluate some important n-ary projectors in linear time $O(n)$.

The technique presented here is independent of the constraint domain. We show that the same technique can be used for indexicals over finite integer domain variables.

Run-time complexity of projectors. The time needed for evaluating a projector depends on the size of its defining range expressions. We define the *size* of a range expression R as the number of set operations (union, intersection, complement) R contains, and write it $|R|$. To evaluate a projector $p_x = (R_1 \subseteq x \subseteq R_2)$, one has to perform $|R_1| + |R_2|$ set operations. Abstracting from the cost of individual operations (as it depends on how sets are implemented), the run-time of p_x is in $O(|R_1| + |R_2|)$.

Example for an n-ary propagator. The finite set constraint $y = \bigcup_{1 \leq i \leq n} x_i$ can be stated as $n+1$ finite set projectors:

$p_y = (\mathrm{glb}(x_1) \cup \cdots \cup \mathrm{glb}(x_n) \subseteq y \subseteq \mathrm{lub}(x_1) \cup \cdots \cup \mathrm{lub}(x_n))$

$p_{x_i} = (\mathrm{glb}(y) \setminus \bigcup_{j \neq i} \mathrm{lub}(x_j) \subseteq x_i \subseteq \mathrm{lub}(y))$ for all x_i

The range expressions in each projector have size n. Propagating all projectors once therefore requires time $O(n^2)$. If however these $n+1$ projectors are grouped, and thus their order of propagation is fixed, we can apply a generalized form of CSE in order to propagate in linear time.

Let us assume we propagate in the order $p_{x_1} \ldots p_{x_n}$. Then at step p_{x_i}, we know that for all $j > i$, we have not yet changed the domain of x_j. Thus, we can use a precomputed table $right[i] = \bigcup_{j>i} \mathrm{lub}(x_j)$. The other half of the union, $left_i = \bigcup_{j<i} \mathrm{lub}(x_j)$, can be maintained incrementally while moving from step $i-1$ to step i. The projectors can thus be written as

$p_{x_i} = (\mathrm{glb}(y) \setminus (right[i] \cup left_i) \subseteq x_i \subseteq \mathrm{lub}(y))$ for all i

Computing the $right[i]$ requires time $O(n)$. Maintaining $left_i$ is constant time, and each resulting projector p_{x_i} can be executed in time $O(1)$, too. This yields $O(n)$ for running all projectors once.

Indexed range expressions. We now extend range expressions so that they can be evaluated efficiently in the n-ary case, using the method sketched in the example above.

We assume that a subset of the variables $Var_{\mathrm{idx}} \subseteq Var$ is *indexed*, such that $x_i \in Var_{\mathrm{idx}}$ for all $1 \leq i \leq k$. We extend range expressions to *indexed range expressions*:

$$R \quad ::= x \mid R \cup R \mid R \cap R \mid \overline{R} \mid \bigcup_{1 \leq j \leq k, j \neq i} x_j \mid \bigcap_{1 \leq j \leq k, j \neq i} x_j \mid \emptyset$$

To simplify presentation, we do not consider nested indexed range expressions here. A family of projectors can now be stated together as $p_{x_i} = (R_1 \subseteq x_i \subseteq R_2)$. We extend the functions r_{lub} and r_{glb} evaluating range expressions to take the index i of the projector as an argument. The functions r_{lub} and r_{glb} implement the optimization sketched above:

$r_{\mathrm{lub}}(\bigcup_{1 \leq j \leq n, j \neq i} x_j, i, S) = right[i] \cup left_i$

where $right$ and $left_i$ are the "two halves" of the union as described in the example. The evaluation of the lower bound and the intersection is analogous.

From specification to indexed range expressions. In order to generate indexed range expressions from formulas as specifications, we have two options. We can either add syntactic sugar to the formulas that allows to express indexed conjunctions and disjunctions, or we search for sub-formulas of the form $\bigwedge_{i \neq j} x_i$ (and similar for disjunction).

Application to n-ary finite domain projectors. The same scheme applies to finite domain projectors. An n-ary linear equation $\sum_i x_i = c$ can be stated as

$$p_{x_i} = x_i \text{ in } c - \textstyle\sum_{j \neq i} \max(x_i) \ \ldots \ c - \sum_{j \neq i} \min(x_i)$$

Again, if the p_{x_i} are evaluated in fixed order, the sums can be precomputed. As for the set projectors, this scheme allows propagation in time $O(n)$ instead of $O(n^2)$.

10 Conclusions and Future Work

We have presented two specification languages: ∃MSO, a high-level, purely declarative language for specifying finite set constraints, and range expressions, a programming language for implementing finite set projectors. Set projectors transfer the ideas of indexicals to the domain of finite sets.

We have captured both languages within one formal framework. On the one hand, this allows us to prove soundness and completeness for projectors with respect to constraints specified as formulas. On the other hand, we can derive sound and complete sets of projectors from constraints specified as formulas. Furthermore, we have shown that we can derive sound propagators for negated and reified constraints, and that ∃MSO is a suitable specification language for BDD-based finite set solvers.

With ∃MSO we thus have an expressive, declarative, high-level specification language for a large class of sound and complete finite set projectors, both for domain approximations using convex sets, and for complete domain representations using BDDs.

The run-time analysis we have presented shows that using plain projectors for n-ary constraints results in quadratic run-time. We have solved this problem with the help of indexed range expressions and evaluating projectors in a group, leading to linear run-time for important n-ary constraints. This result carries over to finite domain indexicals.

An implementation of finite set projectors is available in the Gecode library.

Future work. We are currently integrating a BDD-based solver into the Gecode library. This will allow us to use the same constraint specifications with different solvers. In addition, it will ease the comparison of propagation strength as well as efficiency of different finite set solvers.

The translation from ∃MSO formulas to range expressions still has to be implemented. We currently conduct our experiments by translating formulas by hand.

Besides the implementation, our focus is on extensions of the logic as well as the projector language. We will add cardinality reasoning, which has proven very effective for several areas of application. An interesting further question is whether and how propagators for a domain approximation based on lexicographic bounds [17] can be derived automatically.

Acknowledgements. Christian Schulte is partially funded by the Swedish Research Council (VR) under grant 621-2004-4953. Guido Tack is partially funded by DAAD travel grant D/05/26003. The authors thank Mikael Lagerkvist and the anonymous reviewers for comments on earlier versions of this paper.

References

1. K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.
2. M. Ågren, P. Flener, and J. Pearson. Incremental algorithms for local search from existential second-order logic. In P. van Beek, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, volume 3709 of *LNCS*, pages 47–61. Springer, 2005.
3. N. Beldiceanu, M. Carlsson, and T. Petit. Deriving filtering algorithms from constraint checkers. In Wallace [21], pages 107–122.
4. F. Benhamou. Heterogeneous Constraint Solving. In *Proceedings of the fifth International Conference on Algebraic and Logic Programming (ALP'96), LNCS 1139*, pages 62–76. Springer, 1996.
5. C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, and T. Walsh. The range and roots constraints: Specifying counting and occurrence constraints. In *IJCAI*, pages 60–65, Aug. 2005.
6. B. Carlson. *Compiling and Executing Finite Domain Constraints*. PhD thesis, Uppsala University, Sweden, 1995.
7. B. Carlson, M. Carlsson, and D. Diaz. Entailment of finite domain constraints. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 339–353. MIT Press, 1994.
8. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In H. Glaser, P. H. Hartel, and H. Kuchen, editors, *PLILP*, volume 1292 of *LNCS*, pages 191–206. Springer, 1997.
9. P. Codognet and D. Diaz. Compiling constraints in clp(FD). *Journal of Logic Programming*, 27(3):185–226, 1996.
10. C. Gervet. Interval propagation to reason about sets: Definition and implementation of a practical language. *Constraints*, 1(3):191–244, 1997.
11. P. Hawkins, V. Lagoon, and P. Stuckey. Solving set constraint satisfaction problems using ROBDDs. *J. Artif. Intell. Res. (JAIR)*, 24:109–156, 2005.
12. P. V. Hentenryck, V. A. Saraswat, and Y. Deville. Constraint processing in cc(FD). Technical report, Brown University, 1991.
13. P. V. Hentenryck, V. A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). *Journal of Logic Programming*, 37(1-3):293–316, 1998.
14. Intelligent Systems Laboratory. SICStus Prolog user's manual, 3.12.1. Technical report, Swedish Institute of Computer Science, Box 1263, 164 29 Kista, Sweden, 2006.
15. T. Müller. *Constraint Propagation in Mozart*. Doctoral dissertation, Universität des Saarlandes, Naturwissenschaftlich-Technische Fakultät I, Fachrichtung Informatik, Saarbrücken, Germany, 2001.
16. J.-F. Puget. PECOS: A high level constraint programming language. In *Proceedings of the first Singapore international conference on Intelligent Systems (SPICIS)*, pages 137–142, 1992.
17. A. Sadler and C. Gervet. Hybrid set domains to strengthen constraint propagation and reduce symmetries. In Wallace [21], pages 604–618.
18. C. Schulte and P. J. Stuckey. Speeding up constraint propagation. In Wallace [21], pages 619–633.
19. C. Schulte and G. Tack. Views and iterators for generic constraint implementations. In M. Carlsson, F. Fages, B. Hnich, and F. Rossi, editors, *Recent Advances in Constraints, 2005*, volume 3978 of *LNCS*, pages 118–132. Springer, 2006.
20. The Gecode team. Generic constraint development environment. www.gecode.org, 2006.
21. M. Wallace, editor. *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, volume 3258 of *LNCS*. Springer, 2004.