# Towards Solver-Independent Propagators*

Jean-Noël Monette, Pierre Flener, and Justin Pearson

Uppsala University, Department of Information Technology, Uppsala, Sweden
{jean-noel.monette,pierre.flener,justin.pearson}@it.uu.se

**Abstract.** We present an extension to indexicals to describe propagators for global constraints. The resulting language is compiled into actual propagators for different solvers, and is solver-independent. In addition, we show how this high-level description eases the proof of propagator properties, such as correctness and monotonicity. Experimental results show that propagators compiled from their indexical descriptions are sometimes not significantly slower than built-in propagators of Gecode. Therefore, our language can be used for the rapid prototyping of new global constraints.

## 1 Introduction

One of the main assets of constraint programming (CP) is the existence of numerous filtering algorithms, called propagators, that are tailored specially for global constraints and allow one to solve efficiently hard combinatorial problems. Successful CP solvers enable the definition of new constraints and their associated propagators. However, it may be a tedious task to implement a propagator that is correct, efficient, compliant with the specific interface of the solver, and hand-coded in the solver's implementation language, such as C++.

In this paper, we propose a solver-independent language to describe a large class of propagators. Our contribution towards such a language is twofold.

First, we ease the implementation and sharing of propagators for constraints. The propagators are described concisely and without reference to the implementation details of any solver. Implementations of propagators are generated from their description. This allows one to prototype rapidly a propagator for a new constraint, for which one has no built-in propagator or no (good enough) decomposition; the generated propagator may even serve as a baseline for further refinement. This also allows one to integrate an existing constraint into another solver, as each solver can be equipped with its own back-end for the propagator description language. We believe that such a language can play the same role for sharing propagators between solvers as solver-independent modelling languages play for sharing models between solvers.

Second, we ease the proof of propagator properties, such as correctness and monotonicity. The higher level of abstraction of our propagator description language allows us to design tools to analyse and transform a propagator. This provides a method to apply systematically theoretical results on propagators.

* This work is supported by grant 2011-6133 of the Swedish Research Council (VR). We thank the anonymous reviewers and Christian Schulte for their useful comments.

Our approach is based on the seminal work on indexicals [27]. In a nutshell, an indexical defines a restriction on the domain of a decision variable, given the current domains of other decision variables. Indexicals have been used to implement user-defined constraints in various finite-domain systems, such as SICStus Prolog [6]. While indexicals can originally only deal with constraints of fixed arity, we extend them to deal with constraints of non-fixed arity (often referred to as global constraints) by handling arrays of decision variables and operations on such arrays (iteration and $n$-ary operators). Also, in contrast to classical implementations of indexicals, indexicals are not interpreted here, but compiled into instructions in the source language of the targeted solver.

The paper is structured as follows. After defining the relevant background (Section 2) and presenting motivating examples of propagator descriptions in our language (Section 3), we list our design decisions behind the language (Section 4). Next we describe our language (Section 5), show how to analyse propagators written in it (Section 6), discuss our current implementation (Section 7), and experimentally evaluate it (Section 8). We end the paper with a review of related work and a look at future research directions (Section 9).

## 2 Background

Let $X$ be a set of integer decision variables that take values in some universe $\mathcal{U}$, where $\mathcal{U}$ can be $\mathbb{Z}$ but is in practice a subset thereof. In a finite-domain (FD) solver, a *store* is a mapping $S\colon X \to \mathcal{P}(\mathcal{U})$, where $\mathcal{P}(\mathcal{U})$ is the power set of $\mathcal{U}$. For a variable $x \in X$, the set $S(x)$ is called the *domain* of $x$ and is the set of possible values of $x$. A store $S$ is an *assignment* if every variable has a singleton domain; we say that such variables are *ground*. A store $S$ is *failed* if some variable has an empty domain. A store $S$ is *stronger* than a store $T$ (denoted by $S \sqsubseteq T$) if the domain under $S$ of each variable is a subset of its domain under $T$.

A *constraint* $C(Y)$ on a sequence $Y$ over $X$ is a restriction on the possible values that these variables can take at the same time. The constraint $C(Y)$ is a subset of $\mathcal{U}^n$, where $n$ is the length of $Y$. An assignment $A$ *satisfies* a constraint $C(Y)$ if the sequence $[v \mid \{v\} = A(y) \wedge y \in Y]$ is a member of $C(Y)$. Given a store $S$, a value $v \in S(y)$ for some $y \in Y$ is *consistent* with a constraint $C$ if there exists an assignment $A \sqsubseteq S$ with $A(y) = \{v\}$ that satisfies $C$. A constraint $C$ is *satisfiable* in a store $S$ if there exists an assignment $A \sqsubseteq S$ that satisfies $C$. A constraint $C$ is *entailed* in a store $S$ if all assignments $A \sqsubseteq S$ satisfy $C$.

A *checker* for a constraint $C$ is a function that tells if an assignment satisfies $C$ or not. A *propagator* for a constraint $C$ is a function from stores to stores whose role is to remove domain values that are not consistent (or *inconsistent*) with $C$. We are here interested in writing propagators. In actual solvers, the propagators are not returning a store but modifying the current store. There are a number of desirable properties of a propagator:

- *Correct*: A correct propagator never removes values that are consistent with respect to its constraint. This property is mandatory for all propagators.

- *Checking*: A checking propagator decides if an assignment satisfies its constraint. This divides into *singleton correctness* (accept all satisfying assignments) and *singleton completeness* (reject all non-satisfying assignments).
- *Contracting*: A propagator $P$ is contracting if $P(S) \sqsubseteq S$ for all stores $S$.
- *Monotonic*: A propagator $P$ is monotonic if $S \sqsubseteq T$ implies $P(S) \sqsubseteq P(T)$ for all stores $S$ and $T$.
- *Domain consistent* (*DC*): A DC propagator removes all inconsistent values. Weaker consistencies exist, such as bounds consistency and value consistency.
- *Idempotent*: A propagator $P$ is idempotent if $P(S) = P(P(S))$ for all stores $S$. This allows an improved scheduling by the solver [22].

In addition, and for efficiency reasons, one aims at propagators with a low time complexity. This is the reason why domain consistency is often replaced by weaker consistencies. Another concern is to avoid executing a propagator that cannot remove any value from the current store. Several mechanisms may be implemented by propagators to avoid such executions: report idempotency, report entailment, subscribe only to relevant modification events of the domains.

Indexicals were introduced in [27] to describe propagators for the cc(FD) solver. An *indexical* (*expression*) is of the form $x$ `in` $\sigma$, meaning that the domain of the decision variable $x$ must be restricted to the intersection of its current domain with the set $\sigma$; the set $\sigma$ may depend on other variables, and if so is computed based on their domains in the *current* store. An indexical is executed whenever the domain of one of the variables appearing in $\sigma$ is modified. A propagator is typically described by several indexicals, namely one for each decision variable of the constraint. Indexicals have been included in several other systems, featuring extensions such as checking indexicals [6], conditional expressions [23], [6], and guards [28].

## 3 Examples of Propagator Descriptions

In this paper, we extend the syntax of indexicals to deal with arrays of decision variables and operations on such arrays (iteration and $n$-ary operators). We first present a few examples of propagator descriptions that showcase the main features of our language. This will lead us to explain the decisions we made in the design of the language, and a more precise definition of the language.

Figure 1 presents the $\textsc{Sum}(X, N)$ constraint, which holds if the sum of the values in the array $X$ is equal to $N$. It is possible to describe several propagators for a constraint, and to write an optional checker. In particular, we have here two propagators: `v1` uses the entire domains of the variables (e.g., `dom(N)`), while `v2` only uses their bounds (e.g., `min(N)`). A propagator description is comprised of a set of indexicals. The domains of variables can be accessed using the four functions `dom`, `min`, `max`, and `val`. Arithmetic operators can be applied on integers as well as on sets. The `sum` operator is $n$-ary, as it operates on a sequence of values of arbitrary length. The `rng` operator denotes the range of indices of an array, and $\ell$`..`$u$ the range of integers from $\ell$ to $u$ included. Line 9 reads as: "The domain of `N` must be intersected with the range whose lower bound is the sum of the

```
1  def SUM(vint[] X,vint N){
2    propagator(v1){
3      N in sum(i in rng(X))(dom(X[i]));
4      forall(i in rng(X)){
5        X[i] in dom(N) - sum(j in {k in rng(X):k!=i})(dom(X[j]));
6      }
7    }
8    propagator(v2){
9      N in sum(i in rng(X))(min(X[i])) .. sum(i in rng(X))(max(X[i]));
10     forall(i in rng(X)){
11       X[i] in min(N) - sum(j in {k in rng(X):k!=i})(max(X[j])) ..
12                    max(N) - sum(j in {k in rng(X):k!=i})(min(X[j]));
13     }
14   }
15   checker{ val(N) = sum(i in rng(X))(val(X[i])) }
16 }
```

**Fig. 1.** Code for the SUM constraint, with two propagators

smallest values in the domains of all the variables in `X`, and whose upper bound is the sum of the largest values in the domains of all the variables in `X`." This example also shows how to write loops (`forall` in lines 4–6 and 10–13).

Figure 2 presents the EXACTLY$(X, N, v)$ constraint, which holds if exactly $N$ variables of the array $X$ are equal to the given value $v$. This example illustrates the use of conditions (`when`), boolean-to-integer conversion (`b2i`$(\mathit{false}) = 0$ and `b2i`$(\mathit{true}) = 1$), and reference to other constraints (`EQ` and `NEQ`, constraining a variable to be respectively equal to, and different from, a given value). The functions `entailed` and `satisfiable` check the status of a constraint given the current domains of the variables, while `post` invokes a propagator of the given constraint. Lines 3–4 restrict the domain of `N` to be between two bounds. The lower bound is computed as the number of variables in `X` that *must* be assigned to `v`, and the upper bound as the number of variables that *may* be assigned to `v`. The body of the loop removes `v` from the domain of a variable (lines 6–9), or fixes a variable to `v` (lines 10–13), when some conditions involving the other variables are respected. The domain modifications are performed by invoking the propagation of other constraints.

## 4 Language Design Decisions

The language, as showcased in the previous section and defined more precisely in the next section, has been designed according to the following decisions.

The language is based on indexicals. Indexicals have already been used successfully in several solvers, and a fair amount of work has been done to deal with their use and properties, e.g. in [5] and [9]. Also, indexicals are very simple to understand and are often very close to the first reasoning one might come up

```
1  def EXACTLY(vint[] X, vint N, int v){
2    propagator{
3      N in sum(i in rng(X))(b2i(entailed(EQ(X[i], v)))) ..
4                sum(i in rng(X))(b2i(satisfiable(EQ(X[i], v))));
5      forall(i in rng(X)){
6        once(val(N) <=
7          sum(j in {j in rng(X):i!=j})(b2i(entailed(EQ(X[j], v))))){
8          post(NEQ(X[i], v));
9        }
10       once(val(N) >
11         sum(j in {j in rng(X):i!=j})(b2i(satisfiable(EQ(X[j], v))))){
12         post(EQ(X[i], v));
13       }
14     }
15   }
16   checker{ val(N) = sum(i in rng(X))(b2i(val(X[i]) = v)) }
17 }
```

**Fig. 2.** Code for the EXACTLY constraint

with when thinking about a propagator. One of the restrictions that we currently preserve is that indexicals are *stateless*, hence we cannot describe advanced propagators, such as a DC propagator for the ALLDIFFERENT constraint [19]. This is quite a strong limitation, but a choice must be made between the simplicity of the language and the intricacy of the propagators. However, a large number of constraints have efficient enough stateless propagators.

The language is strongly typed, in order to simplify the understanding and compilation of propagators. This requires the addition of the `b2i` operator.

We introduce arrays and $n$-ary operators to deal effectively with global constraints. For example, the expression `sum(i in rng(X))(val(X[i]))` is parametrised by the looping index (`i` here), its domain (`rng(X)` here), and the index-dependent expression (`val(X[i])` here) that must be aggregated (summed here).

We also introduce meta-constraints, constraint invocation, and local variables in order to help write concise propagators. See Section 5 for details.

For simplicity of use, we want only a few different operators and language constructs. This also allows us to have a relatively simple compilation procedure. However it is not impossible that some new constructs will be added in the future, but with care.

For generality (in the FD approach), we refrain from adding solver-specific hooks. In particular, our language only has four accessors (see Section 5) to the domain of a variable. The other communication channels with a solver are domain narrowing functions, which are provided by any FD solver, and a fail mechanism (which can be mimicked by emptying a domain).

Our language is also missing some constructs that are found in the implementation of constraints, such as entailment detection, watched literals [13], and

fine-grained events [15]. Part of our future work will be dedicated to studying how these can be included without overcomplicating the language.

## 5 Definition of the Language

In Section 5.1, we define the syntax and semantics of our language. It is strongly typed and has five basic types: integers (`int`), booleans (`bool`), sets of integers (`set`), integer decision variables (`vint`), and constraints (`cstr`). This last type is discussed in Section 5.2. We support arrays of any basic type (but currently not arrays of arrays). Identifiers of (arrays of) decision variables start with an uppercase letter. Identifiers of constants denoting integers, booleans, sets, and arrays thereof start with a lowercase letter.

### 5.1 Syntax and Semantics

Figure 3 presents the grammar of our language. We now review the different production rules. The main rule (`CSTR`) defines a constraint. A *constraint* is defined by its name and list of arguments. A constraint definition also contains the description of one or more propagators and an optional checker.

A *propagator* has an optional identifier and contains a list of instructions. An instruction (`INSTR`) can be an *indexical*, $x$ `in` $\sigma$, whose meaning is that the domain of the decision variable $x$ must be restricted to the intersection of its current domain with the set $\sigma$. Other instructions are `fail` and `post`. The effect of `fail` is to transform the current store into a failed store. The instruction `post(`$C$`,`$P$`)` invokes the propagator $P$ of constraint $C$; if $P$ is not specified, then the first (or only) propagator of $C$ is invoked. There are two control structures. The `forall` control structure creates an iteration over a set, and `once` creates a conditional block; the reason for not naming the latter `if` is to stress that propagators should be monotonic, and that once the condition becomes true, it should remain true. See Section 6 for a discussion on monotonicity.

Most of the rules on sets, integers, and booleans do not need any explanations or were already explained in Section 3. Some constants are defined: `univ` denotes the universe $\mathcal{U}$, `inf` its infimum, `sup` its supremum, and `emptyset` the empty set. Arithmetic operations on integers are lifted as point-wise operations to sets.

There are four accessors to the domain of a decision variable: `dom(`$x$`)`, `min(`$x$`)`, `max(`$x$`)`, and `val(`$x$`)` denote respectively the domain of decision variable $x$, its minimum value, its maximum value, and its unique value. As `val(`$x$`)` is only determined when the decision variable $x$ is ground, the compiler must add guards to ensure a correct treatment when $x$ is not ground.

While the instruction `post` invokes the propagator of another constraint, the functions `entailed`, `satisfiable`, and `check` query the status of another constraint. Let $S$ be the current store: `entailed(`$c$`)` and `satisfiable(`$c$`)` decide whether the constraint $c$ is entailed (respectively, satisfiable) in $S$; if $S$ is an assignment, then the function `check(`$c$`)` can be called and decides whether $S$ satisfies the constraint $c$ (an example will be given in the next sub-section).

```
CSTR    ::= def CNAME(ARGS){ PROPAG+ CHECKER?)}
PROPAG  ::= propagator(PNAME?){ INSTR* }
CHECKER ::= checker{ BOOL }
INSTR   ::= VAR in SET ; | post(CINVOKE,PNAME?); | fail; |
            once(BOOL){ INSTR* } | forall(ID in SET){ INSTR* }
SET     ::= univ | emptyset | ID | INT..INT | rng(ID) | dom(VAR) |
            NSETOP(ID in SET)(SET) | -SET | SET BSETOP SET |
            {INT+} | {ID in SET:BOOL}
INT     ::= inf | sup | NUM | ID | card(SET)| min(SET) | max(SET) |
            min(VAR) | max(VAR) | val(VAR) | - INT | INT BINTOP INT |
            b2i(BOOL) | NINTOP(ID in SET)(INT)
BOOL    ::= true | false | ID | INT INTCOMP INT | INT memberOf SET |
            SET SETCOMP SET | not BOOL | BOOL BBOOLOP BOOL |
            NBOOLOP(ID in SET)(BOOL) |
            entailed(CINVOKE) | satisfiable(CINVOKE) | check(CINVOKE)
BINTOP  ::= + | - | * | / | mod
NINTOP  ::= sum | min | max
BSETOP  ::= union | inter | minus | + | - | * | / | mod
NSETOP  ::= union | inter | sum
INTCOMP ::= = | != | <= | < | >= | >
SETCOMP ::= = | subseteq
BBOOLOP ::= and | or | =
NBOOLOP ::= and | or
CINVOKE ::= CNAME | CNAME(ARGS)
```

**Fig. 3.** BNF-like grammar of our language. Constructions in grey were already in previous definitions of indexicals [27], [6]. The rules corresponding to `ARGS` (list of arguments), `CNAME`, `PNAME`, `ID`, `VAR` (respectively identifier of a constraint, propagator, constant, and variable), and `NUM` (integer literal) are not shown.

### 5.2 Meta-constraints

A new feature of our language is what we call a meta-constraint, which is a constraint that takes other constraint(s) as argument(s). Meta-constraints allow one to write more concise propagators by encapsulating common functionalities.

For example, the AMONG$(X, N, s)$ constraint, which holds if there are $N$ elements in array $X$ that take a value in set $s$, would be described almost identically to the EXACTLY$(X, N, v)$ constraint in Figure 2. The common code can be factored out in the meta-constraint `COUNT(vint[] X, vint N, cstr C, cstr NC)`, whose full description is not shown here, but whose meaning is defined by its checker: `val(N) = sum(i in rng(X))(b2i(check(C(X[i]))))`, that is exactly `N` variables of the array `X` satisfy constraint `C`. The argument `NC` is the negation of constraint `C` (see [3] for how to negate even global constraints), and is used in the propagator of COUNT (in the way `NEQ` is used on line 8 of Figure 2). We can then describe EXACTLY and AMONG as shown in Figure 4. The COUNT meta-constraint is closely related to the cardinality operator [25] but we allow the user to describe more meta-constraints.

```
def EXACTLY(vint[] X, vint N, int v){
  propagator{
    cstr EQv(vint V):= EQ(V,v);
    cstr N_EQv(vint V):= NEQ(V,v);
    post(COUNT(X,N,EQv,N_EQv));
  }
}
```

```
def AMONG(vint N, vint[] X, set s){
 propagator{
  cstr INs(vint V):= INSET(V,s);
  cstr NINs(vint V):= NOTINSET(V,s);
  post(COUNT(X,N,INs,NINs));
 }
}
```

**Fig. 4.** EXACTLY and AMONG, described using the COUNT meta-constraint

## 6 Syntactic Analysis and Tools

One of our objectives is to ease the proof of propagator properties. Before turning to the actual compilation, we show how our language helps with this, and with other propagator-writing related functionalities.

### 6.1 Analysis

Among the properties of a propagator presented in Section 2, most are difficult to prove for a given propagator (except contraction, which indexicals satisfy by definition). However, as has been shown in [5], it is possible to prove the monotonicity of indexicals. This result can be lifted to our more general language. We show further how to prove the correctness of some propagators with respect to their constraints.

*Monotonicity.* The procedure to check the monotonicity of indexicals is combined with the addition of guards for `val` accesses. The syntax tree representing the indexicals is traversed by a set of mutually recursive functions. To ensure monotonicity of the whole propagator, each function verifies an *expected* behaviour of the subtree it is applied on. The recursive functions are labelled *monotonic*, *anti-monotonic*, and *fixed* for boolean expressions; *increasing*, *decreasing*, and *fixed* for integer expressions; *growing*, *shrinking*, and *fixed* for set expressions; and *monotonic* for instructions. For instance, the *increasing* function verifies that its integer expression argument is non-strictly increasing when going from a store to a stronger store. These functions return two values: whether the expression is actually respecting its expected behaviour, and the set of variables that need to be ground to ensure a safe use of the `val` accessor. This set of variables is used to add guards in the generated propagator. Those guards are added not only to instructions, but also inside the body of `b2i` expressions.

For lack of space, we cannot exhibit all the rules that make up the recursive functions (there are about 200 rules). Instead we show a few examples. As an example of a rule, consider the call of the function *increasing* on an expression of the form `min(`$i$ `in` $\sigma$`)(`$e$`)`. For this expression to be increasing, $\sigma$ must be *shrinking* and $e$ must be *increasing*. In addition, the set of variables to guard is the union of the variables that must be ground for those two subexpressions.

The table below shows some examples of the results of the procedure. In this table, `ground(`$x$`)` represents an operator (not part of our indexical language) that decides if variable $x$ is ground. The second column shows where guards are added to indexicals. The third column reports if the indexical is proven monotonic or not.

| Original expression | Guarded expression | Mono |
|---|---|---|
| `X in {val(Y)}` | `once(ground(Y)) X in {val(Y)}` | true |
| `B in b2i(val(X)=v) .. b2i(v memberOf dom(X))` | `B in b2i(ground(X) and val(X)=v) .. b2i(v memberOf dom(X))` | true |
| `once(min(B)=1) X in {v}` | `once(min(B)=1) X in {v}` | false |
| `once(min(B)>=1)X in {v}` | `once(min(B)>=1) X in {v}` | true |
| `once(val(B)=1) X in {v}` | `once(ground(B) and val(B)=1)X in {v}` | true |

The first line just adds a guard to the indexical. The second line shows that a guard can be added inside a `b2i` expression so that it returns `0` while `X` is not ground. The three last lines show how small variations change the monotonicity of an expression. The condition `min(B)=1` is (syntactically) not monotonic, because in general this condition might be true in some store and become false in a stronger store; however if we know that `B` represents a boolean (with domain `0..1`), then we can replace the equality by an inequality as done in the fourth example. The last line shows another way to get monotonicity, namely by replacing the `min` accessor by `val`; this requires the addition of a guard.

The soundness of the monotonicity checking procedure can be shown by induction on the recursive rules, as suggested in [5]. However, as this procedure is syntactical, it is incomplete. An example of propagator for $\textsc{Eq}(X, Y)$ that is monotonic but not recognised as such is given on the left of Figure 5. The procedure does not recognise the monotonicity because it requires the sets over which the `forall` loops iterate to be growing (because the indexical expressions that are applied on a store must also be applied on a stronger store), while `dom(`$x$`)` can only shrink. However, this is not the simplest way to describe propagation for this constraint: a 2-line propagator can be found on the upper right of Figure 5.

*Correctness.* To prove algorithmically that a propagator is correct with respect to its constraint, we use the known fact that a propagator is correct if it is singleton-correct and monotonic. We devise an incomplete but sound procedure to prove that a propagator $P$ is singleton-correct with respect to its checker $C$, and hence with respect to its constraint (assuming the checker is correct). We need to prove that if an assignment satisfies $C$, then it is not ruled out by $P$. To this end, from the indexical description of $P$, we derive a formula $C(P)$ that defines which assignments are accepted by the propagator. Singleton correctness then holds if the formula $C \wedge \neg C(P)$ is unsatisfiable (i.e., if $C \Rightarrow C(P)$). To derive $C(P)$, we transform the indexicals into an equivalent checking formula using the following rewrite rules:

```
1  def EQ(vint X1, vint X2){
2    propagator{
3      forall(i in dom(X1)){
4        once(not i memberOf dom(X2)){
5          X1 in univ minus {i};
6        }
7      }
8      forall(i in dom(X2)){
9        once(not i memberOf dom(X1)){
10         X2 in univ minus {i};
11       }
12     }
13   }
14 }
```

```
1  def EQ(vint X, vint Y){
2    propagator{
3      X in dom(Y);
4      Y in dom(X);
5    }
6    checker{ val(X) = val(Y) }
7  }
```

```
1  def EQ(vint X, int cY){
2    propagator{
3      X in {cY};
4      once(not cY
5           memberOf dom(X))
6        fail;
7    }
8    checker{ val(X) = cY }
9  }
```

**Fig. 5.** Variations of the EQ constraint

$\texttt{dom}(x) \rightarrow \{\texttt{val}(x)\}$
$\texttt{min}(x) \rightarrow \texttt{val}(x)$
$\texttt{max}(x) \rightarrow \texttt{val}(x)$
$\texttt{fail} \rightarrow \texttt{false}$

$x\ \texttt{in}\ \sigma \rightarrow \texttt{val}(x)\ \texttt{memberOf}\ \sigma$
$\texttt{once}(b)\{y\} \rightarrow (\texttt{not}\ b)\ \texttt{or}\ y$
$\texttt{forall}(i\ \texttt{in}\ \sigma)\{y\} \rightarrow \texttt{and}(i\ \texttt{in}\ \sigma)(y)$

Our current procedure to prove the unsatisfiability of $C \wedge \neg C(P)$ tries to simplify the formula to `false` using rewrite rules. We implemented around 240 rewrite rules, ranging from boolean simplification (e.g., $\texttt{false or}\ b \rightarrow b$) to integer and set simplification (e.g., $\texttt{min}(i\ \texttt{in}\ \ell..u)(i) \rightarrow \ell$) and partial evaluation (e.g., $2 + x + 3 \rightarrow x + 5$). As this procedure is incomplete and able to prove singleton correctness only for a small portion of the propagators (see Section 7.2), we plan to improve it by calling an external prover.

As an example, applying the propagator-to-checker transformation on the propagator on the upper right of Figure 5 results in the formula `val(X) memberOf {val(Y)} and val(Y) memberOf {val(X)}`. This formula can be shown equivalent to the checker of the constraint (using the following rules: $x$ `memberOf` $\{y\} \rightarrow x$ `=` $y$, $b$ `and` $b \rightarrow b$, and $b$ `and not` $b \rightarrow$ `false`). In summary, this propagator can be automatically proven monotonic, singleton correct, singleton complete (see below), and therefore correct and checking.

*Checking.* The approach to proving correctness can also be used to prove that a propagator is checking. Indeed, singleton completeness is shown by proving the implication $C(P) \Rightarrow C$ (the converse of singleton correctness), and a propagator that is singleton-correct and singleton-complete is checking (i.e., $C(P) \Leftrightarrow C$).

### 6.2 Transformation

In addition to the analysis, we can algorithmically transform a propagator. We have devised two first transformations that seem of interest: changing the level of reasoning, and grounding some decision variables.

*Changing the level of reasoning.* As shown in the SUM example (Figure 1), a propagator can be described to use different levels of reasoning according to the amount of data it uses:

- Under domain reasoning, the whole domains of decision variables may be used to perform propagation.
- Under bounds reasoning, only the bounds of the domains are used (i.e., the `dom` accessor does not appear).
- Under value reasoning, no propagation is performed until some variables are ground (i.e., only the `val` accessor is used).

A few remarks are necessary here. First, the level of reasoning can be distinct for the different variables of a constraint. Second, those levels of reasoning are not directly linked to the usual notions of consistency (domain, bounds, or value consistency). Indeed, using the `dom` accessor does not provide any guarantee of domain consistency. Conversely some propagators that do not use `dom` may achieve domain consistency. Third, the level of reasoning is also distinct from the level of narrowing of variables, which is how propagation affects the domain of variables, i.e., if it only updates the bounds, or if it can create holes.

It is possible to change the level of reasoning from a strong level to a weaker one, i.e., from domain reasoning to bounds reasoning, and from bounds reasoning to value reasoning. All one has to do is to replace the appearances of `dom(`$x$`)` by `min(`$x$`)..max(`$x$`)`, and of `min(`$x$`)` and `max(`$x$`)` by `val(`$x$`)`. This allows one to describe one propagator, and have for free up to three different implementations that one may try and compare.

*Variable grounding.* Another propagator transformation is the grounding of some variables, that is the replacement of a variable argument by a constant (or of an array of variables by an array of constants). Again, this allows one to describe only one propagator for the general case and then specialise it to specific cases. For instance, one might want to derive a propagator for the `EQ(vint,int)` constraint from the one of `EQ(vint,vint)`. The transformation is close in spirit to the computation of the equivalent checking formula of a propagator presented in Section 6.1. The transformation is however only applied to one variable (the one being grounded). The most interesting rewrite rule is that if variable $x$ is replaced by a constant $c$, then $x$ `in` $\sigma$ is replaced by `once(not` $c$ `memberOf` $\sigma$`) fail`. Most of the time this check is redundant with the other instructions of the propagator (as in the example on the lower right of Figure 5, where lines 4–6 check a condition enforced by line 3). However, we have not found yet a general and cheap way to tell when this instruction is indeed redundant. Currently, it is the responsibility of the user to remove it if he wants to.

## 7 Compilation

We now discuss our compiler design decisions and our current compiler.

### 7.1 Compiler Design Decisions

Instead of interpretation, we made the decision to compile our language into the language in which propagators are written for a particular solver. This has the double advantage of having an infrastructure that is relatively independent of the solvers (only the code generation part is solver-dependent), and of having compiled code that is more efficient than interpreted code. The generated code is also self-contained (it can be distributed without the compiler).

The compiled propagators are currently stateless (as are indexical propagators) and use coarse-grained wake-up events. This choice is meant to simplify the compilation. However, upon a proper analysis, it should be possible to produce propagators that incorporate some state or use more fine-grained events.

The compilation produces one propagator for each `propagator` description. The compilation does not alter the order of the indexicals inside a propagator. Furthermore, to get idempotency, the full propagator is repeated until it reaches its internal fixpoint. Another valid choice would have been to create a propagator for each indexical, and let the solver perform the scheduling. We have not evaluated all the potential trade-offs of this choice. An intermediate approach would be to analyse the internal structure of the propagator description to generate a good scheduling policy of the indexicals inside the propagator. This requires substantially more work and is left as future work.

The invocations of propagators (`post`) or checkers (`check`) are replaced by the corresponding code. This is similar to function inlining in classical programming languages. For the functions `entailed` and `satisfiable`, the description of the corresponding checker is first transformed in order to deal with stores that are not assignments. This is done by about 130 rewrite rules forming a recursive procedure similar to the one for monotonicity checking. As a difference, the `val` accessors are replaced by `min`, `max`, or `dom` when possible, or are properly guarded otherwise. For example, calling the *shrinking* function on the singleton {`val(`$x$`)`} returns `dom(`$x$`)`, but calling the *growing* function adds a guard instead. Entailment requires a *monotonic* boolean formula, and satisfiability an *antimonotonic* formula. For example, the generated satisfiability checker of `EQ(X,Y)` is `not dom(X) inter dom(Y) = emptyset`. In turn, the generated entailment checker of `EQ(X,Y)` is `ground(X) and ground(Y) and val(X)=val(Y)`.

The choice of inlining constraint invocations greatly simplifies the compilation process. However, this means that all referenced constraints must be described by indexicals. We plan to explore how we can remove this limitation in order to be able to invoke propagators built into the targeted solver.

### 7.2 Implementation and Target Solvers

We have written a prototype compiler in Java. It uses Antlr [18] for the parsing and StringTemplate [17] for the code generation. Currently, we compile into

propagators for Comet [10], Gecode [12], and Scampi [21]. A big part of the compilation process amounts to rewriting the $n$-ary operators as loops. Some optimisations are performed, such as a dynamic programming pre-computation of arrays (replacing nested loops by successive loops) [24, Section 9] and the factorisation of repeated expressions. The compiler detects the events that should wake up the propagator; this is performed by walking the syntax tree and gathering the variable accessors. The compiler also adds entailment detection to the propagator of constraint $c$, by testing if `entailed(`$c$`)` is true.

Currently, we have written about 700 lines of indexicals for describing 76 propagators of 48 constraints, of which 14 are meta-constraints, 17 are global constraints, and 17 are binary or ternary constraints. Out of the 76 propagators, 69 are proven monotonic, of which 16 are proven singleton-correct and 29 are proven singleton complete, making 16 propagators provably correct. These numbers could be improved with a better unsatisfiability proof procedure.

To get an idea of the conciseness of the language, note that our compiler produces from the 17-line description of Exactly in Figure 2 a propagator for Comet that is about 150 lines of code, and one for Gecode of about 170 lines. We estimate the code for the built-in propagator of this constraint to be around 150 lines of code in Gecode.

The current prototype, as well as the propagator descriptions, are available on demand from the first author.

## 8 Experimental Evaluation

To assess that propagators described by indexicals behave reasonably well, we compare a few generated propagators with built-in propagators of Gecode and simple constraint decompositions. We do not expect the generated propagators to be as efficient as the hand-crafted ones, but the goal is to show that they are a viable alternative when one has little time to develop a propagator for a constraint.

Our experimental setting is as follows. We use Gecode 3.7.3. For each constraint, we search for all its solutions. We repeat the search using several branching heuristics to try and exercise as many parts of the propagators as possible.

The studied constraints are Sum, Maximum, Exactly, and Element. Their indexical descriptions are representative of the other constraints we implemented. In addition, they share the property that one of the variables is functionally dependent on the other ones. This allows us to compare the different propagators of a constraint with a dummy problem where the constraint is absent but the functionally dependent variable is instead fixed to an arbitrary value. As the considered constraint is only defining the functional dependency, the number of solutions is the same and the size of the search tree is the same, but the time spent by propagation is null. We can then compute the runtime of a propagator by subtracting the total runtimes.

For Sum and Maximum, we use bounds-reasoning versions of the indexicals and built-in propagators; for Exactly and Element, we use domain-reasoning

**Table 1.** Relative runtimes (in percent)

| | MAXIMUM | SUM | EXACTLY | ELEMENT |
|---|---|---|---|---|
| Built-in | 100 | 100 | 100 | 100 |
| Indexicals | 125 | 269 | 252 | 118 |
| Decomposition | 195 | 296 | 313 | 204 |
| Automaton | 675 | n/a | n/a | 487 |

versions. The indexical descriptions of SUM and EXACTLY are shown in Figures 1 (`v2`) and 2 respectively. For space reasons, MAXIMUM and ELEMENT are not shown. The decompositions of $\text{SUM}(X, N)$ and $\text{MAXIMUM}(X, N)$ introduce an array $A$ of $n = \|X\|$ auxiliary variables. The decomposition of SUM is expressed as $A[1] = X[1] \wedge \forall_{i \in 2..n} (A[i-1] + X[i] = A[i]) \wedge A[n] = N$, and the one of MAXIMUM is $A[1] = X[1] \wedge \forall_{i \in 2..n} (\max(A[i-1], X[i]) = A[i]) \wedge A[n] = N$. The decomposition of $\text{EXACTLY}(X, N, v)$ introduces an array $B$ of boolean variables and is defined as $\forall_{i \in 1..n} (B[i] \equiv X[i] = v) \wedge N = \sum_{i \in 1..n} B[i]$; the sum of boolean variables is implemented by a built-in propagator. The $\text{ELEMENT}(X, Y, Z)$ constraint (holding if $X[Y] = Z$) is decomposed into $Y \in 1..n \wedge \forall_{i \in 1..n} (Y = i \Rightarrow X[i] = Z)$. Additionally, we use the automaton formulations of MAXIMUM and ELEMENT, given in the Global Constraint Catalogue [4]. The automata of MAXIMUM and ELEMENT and the decomposition of ELEMENT do not perform all the possible pruning (while the other decompositions do so, at least under the used heuristics). This incurs an overhead of about 15% more nodes visited for the automata, and 7% for the decomposition of ELEMENT.

Table 1 presents the relative runtimes of the different implementations of the constraints for arrays of 9 variables over domains of 9 values. The used search heuristics are some combinations of the variable ordering (order of the arguments of the constraints, and within an array the smallest or the largest domain first) and of the value ordering (assign the minimum, split in two, assign the median). For each constraint and each propagator, the runtimes are summed over the different search heuristics. Then the sum of the times to explore the search tree is subtracted. Finally, for each constraint, the sum of the times of each propagator is divided by the sum of the times of the Gecode built-in propagator. Compared to the built-in propagators, the generated propagators induce only a small overhead for MAXIMUM and ELEMENT, but do not behave so well for SUM and EXACTLY. However, in all cases, the indexicals have a better runtime than the decomposition, though sometimes only slightly. In particular, for the EXACTLY constraint, the decomposition has a better runtime for some smaller instances (not shown). We explain the behaviour of the indexicals on this constraint by the fact that it is awakened each time the domain of a variable changes, even though some variables might not affect the constraint status anymore. The built-in propagator and the decomposition are more clever and ignore the variables that cannot take the given value anymore. The indexicals have a much better runtime than the automata.

The propagators compiled from indexicals have an average runtime per call that (necessarily) increases linearly with the number of variables. The runtimes of the built-in propagators increase also but with a much gentler slope.

These experiments show that indexical descriptions of global constraints are useful, but that there is still room for improvement in the compilation.

## 9 Conclusion

We have presented a solver-independent language to describe propagators. The aim is to ease the writing and sharing of propagators and to make proving their formal properties much easier. The resulting language, based on indexicals, is high-level enough to abstract away implementation details and to allow some analyses and transformations. It is compiled into source code for target solvers.

The idea of letting the user write his own propagators is not new. The system cc(FD) [27] is one of the first to have proposed this, through the use of indexicals. Since then, most CP solvers claim to be open, in the sense that any user can add new propagators (especially for global constraints) to the kernel of built-in propagators. In such systems, the user writes code in the host programming language of the solver and integrates it with the solver through an interface defining mainly how to interact with the variables and the core of the solver. Some solvers take a quite different approach and propose a language to define propagation, examples include constraint handling rules [11] and action rules [28]. Our language is a level of abstraction above those approaches, as it can be translated into code for *any* solver. From this point of view, it is close in spirit to what a solver-independent modelling language is to solvers: a layer above solvers to describe easily problem models, which are then compiled into the solver input language. In our case, propagators are described, not models.

Propagators have also been described using atomic constraints and propagation rules [8]. However, that description language has been devised to reason about propagators, and it is not practical for actually implementing propagators in FD solvers, except for approaches based on SAT solvers that use such low-level constraints (e.g., lazy-clause generation [16]).

The pluggable constraints of [20] also decouple the implementation of constraints from the solver architecture, using a solver-independent interface for the communication between the two components. Our approach aims at a higher level of abstraction, possibly losing some fine control.

This paper opens several interesting research directions, in addition to those already listed in Sections 4 and 7.1 for overcoming initial design decisions: the language needs to allow better control of the propagation algorithm while staying simple and general. Simplicity is important, as we believe that having *automated* propagator analysis tools eases the writing of propagators.

Future compilation targets are other FD solvers (e.g., Choco [7] and JaCoP [14]), lazy-clause generators in SAT [16], cutting plane generators in MIP [1], and penalty functions or invariants in constraint-based local search [26], [2].

## References


1. Achterberg, T.: SCIP: Solving constraint integer programs. Mathematical Programming Computation 1, 1–41 (2009)
2. Ågren, M., Flener, P., Pearson, J.: Inferring variable conflicts for local search. In: Benhamou, F. (ed.) Proceedings of CP'06. LNCS, vol. 4204, pp. 665–669. Springer-Verlag (2006)
3. Beldiceanu, N., Carlsson, M., Flener, P., Pearson, J.: On the reification of global constraints. Tech. Rep. T2012:02, Swedish Institute of Computer Science (February 2012), available at http://soda.swedish-ict.se/view/sicsreport/
4. Beldiceanu, N., Carlsson, M., Rampon, J.X.: Global constraint catalog, 2nd Edition (revision a). Tech. Rep. T2012:03, Swedish Institute of Computer Science (February 2012), available at http://soda.swedish-ict.se/view/sicsreport/
5. Carlson, B., Carlsson, M., Diaz, D.: Entailment of finite domain constraints. In: Proceedings of ICLP'94. pp. 339–353. MIT Press (1994)
6. Carlsson, M., Ottosson, G., Carlson, B.: An open-ended finite domain constraint solver. In: Glaser, H., Hartel, P., Kuchen, H. (eds.) Proceedings of PLILP'97. LNCS, vol. 1292, pp. 191–206. Springer-Verlag (1997)
7. CHOCO: An open source Java CP library, http://www.emn.fr/z-info/choco-solver/
8. Choi, C.W., Lee, J.H.M., Stuckey, P.J.: Removing propagation redundant constraints in redundant modeling. ACM Transactions on Computational Logic 8(4) (2007)
9. Dao, T.B.H., Lallouet, A., Legtchenko, A., Martin, L.: Indexical-based solver learning. In: Proceedings of CP'02. LNCS, vol. 2470, pp. 541–555. Springer-Verlag (2002)
10. Dynadec, Dynamic Decision Technologies Inc.: Comet tutorial, v2.0 (2009), http://dynadec.com/
11. Frühwirth, T.W.: Theory and practice of constraint handling rules. Journal of Logic Programming 37(1–3), 95–138 (1998)
12. Gecode Team: Gecode: A generic constraint development environment (2006), available from http://www.gecode.org/
13. Gent, I.P., Jefferson, C., Miguel, I.: Watched literals for constraint propagation in minion. In: Proceedings of CP'06. LNCS, vol. 4204, pp. 182–197 (2006)
14. JaCoP: Java constraint programming solver, http://jacop.osolpro.com/
15. Mohr, R., Henderson, T.: Arc and path consistency revisited. Artificial Intelligence 28, 225–233 (1986)
16. Ohrimenko, O., Stuckey, P., Codish, M.: Propagation via lazy clause generation. Constraints 14, 357–391 (2009)
17. Parr, T.J.: Enforcing strict model-view separation in template engines. In: Proceedings of the 13th international conference on the World Wide Web. pp. 224–233. ACM (2004)
18. Parr, T.J.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Bookshelf (2007)
19. Régin, J.C.: A filtering algorithm for constraints of difference in CSPs. In: Hayes-Roth, B., Korf, R.E. (eds.) Proceedings of AAAI'94. pp. 362–367. AAAI Press (1994)
20. Richaud, G., Lorca, X., Jussien, N.: A portable and efficient implementation of global constraints: The tree constraint case. In: Proceedings of CICLOPS 2007. pp. 44–56 (2007)

21. Scampi (2011), https://bitbucket.org/pschaus/scampi/
22. Schulte, C., Stuckey, P.J.: Speeding up constraint propagation. In: Proceedings of CP'04. LNCS, vol. 3258, pp. 619–633. Springer-Verlag (2004)
23. Sidebottom, G., Havens, W.S.: Nicolog: A simple yet powerful cc(FD) language. Journal of Automated Reasoning 17, 371–403 (1996)
24. Tack, G., Schulte, C., Smolka, G.: Generating propagators for finite set constraints. In: Proceedings of CP'06. LNCS, vol. 4204, pp. 575–589. Springer-Verlag (2006)
25. Van Hentenryck, P., Deville, Y.: The cardinality operator: A new logical connective for constraint logic programming. In: Proceedings of ICLP'91. pp. 745–759 (1991)
26. Van Hentenryck, P., Michel, L.: Differentiable invariants. In: Benhamou, F. (ed.) Proceedings of CP'06. LNCS, vol. 4204, pp. 604–619. Springer-Verlag (2006)
27. Van Hentenryck, P., Saraswat, V., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(FD). Tech. Rep. CS-93-02, Brown University, Providence, USA (January 1993), revised version in *Journal of Logic Programming* 37(1–3):293–316, 1998. Based on the unpublished manuscript *Constraint Processing in cc(FD)*, 1991.
28. Zhou, N.F.: Programming finite-domain constraint propagators in action rules. Theory and Practice of Logic Programming 6, 483–507 (September 2006)