Propagation Engine Prototyping with a Domain Specific Language

Charles Prud'Homme, Xavier Lorca, Rémi Douence, Narendra Jussien

Noname manuscript No.
(will be inserted by the editor)

Propagation Engine Prototyping with a Domain Specific Language

Charles Prud'homme · Xavier Lorca · Rémi Douence · Narendra Jussien

Received: date / Accepted: date

Abstract Constraint propagation is at the heart of constraint solvers. Two main trends co-exist for its implementation: variable-oriented propagation engines and constraint-oriented propagation engines. Those two approaches ensure the same level of local consistency but their efficiency (computation time) can be quite different depending on the instance solved. However, it is usually accepted that there is no best approach in general, and modern constraint solvers implement only one. In this paper, we would like to go a step further providing a solver independent language at the modeling stage to enable the design of propagation engines. We validate our proposal with a reference implementation based on the Choco solver and the MiniZinc constraint modeling language.

Keywords Propagation · Constraint solver · Domain Specific Language · Implementation

C. Prud'homme
EMNantes, INRIA TASC, CNRS LINA,
FR-44307 Nantes Cedex 3, France
Tel.: +33-251-858-368
E-mail: Charles.Prudhomme@mines-nantes.fr

X. Lorca
EMNantes, INRIA TASC, CNRS LINA,
FR-44307 Nantes Cedex 3, France
Tel.: +33-251-858-232
E-mail: Xavier.Lorca@mines-nantes.fr

R. Douence
EMNantes, INRIA ASCOLA, CNRS LINA,
FR-44307 Nantes Cedex 3, France
Tel.: +33-251-858-215
E-mail: Remi.Douence@mines-nantes.fr

N. Jussien
EMNantes, INRIA TASC, CNRS LINA,
FR-44307 Nantes Cedex 3, France
Tel.: +33-251-858-202
E-mail: Narendra.Jussien@mines-nantes.fr

1 Introduction

Constraint propagation which lies at the heart of constraint programming solvers has been extensively studied during the last decades [1, 16]. Several fix-point reasoning algorithms exist [10]; they are mainly represented by modern variants of arc-consistency algorithms [1, 2, 4]. For a given algorithm, several ways of propagating constraints exist ; they are named *orientations* in the following. The orientation defines the information to store in order to propagate domain reductions through the constraint network, *e.g.*, modified variables or constraints likely to provide filtering. The most widely used orientations are based on variable orientation or constraint orientation. For instance, IBM CPO [21], Choco [6], Minion [11] and or-tools [24] use a variable-oriented algorithm, while Gecode [22], SICStus Prolog [5] and JaCoP [23] use a constraint-oriented algorithm. The design of a propagation engine, and more generally a constraint solver, requires choices and compromises in order to combine adaptability (to solve a wide range of problems) with efficiency (to solve them efficiently). Unfortunately, modifying such a central element is challenging. Constraint propagation strategies are tightly related to the solver implementation. It may be a tedious task to implement an algorithm that is correct, efficient, compliant with the specific interface of the solver, and coded in the solver's implementation language. Presumably, only the developers of a solver can safely modify the propagation mechanism. Consequently, propagation engines tend to become monolithic and relatively closed at the modeling stage. Then, even if giving access to constraint propagation algorithms has no purpose for a beginner, it has a strong motivation for an advanced modeler or a tool developer. The former may want to optimize the resolution process without an in-depth knowledge about the underlying solver. The latter may want to rapidly prototype new propagation strategies before a complex phase of internal development.

In this paper, we propose a solver-independent language able to configure constraint propagation at the modeling stage. Our contribution is threefold. First, we present a Domain Specific Language (DSL) to ease constraint propagation engine configuration. We show that it enables expressing a large variety of existing propagation strategies. Second, we exploit the basic properties of our DSL in order to ensure both completeness and correctness of the produced propagation engine. Third, we present a concrete implementation of our DSL based on Choco and an extension of the MiniZinc [20] language. Finally we show that the overhead induced by the DSL and its implementation is operationally acceptable for prototyping propagation engines.

In the following, Section 2 recalls the fundamentals of constraint propagation, as well as state-of-the-art improvements in constraint propagation algorithms. Section 3 is dedicated to the description of the DSL. Section 3.1 defines the set of required techniques and services the underlying solver has to implement to fully support the DSL. Section 3.2 presents and discusses the details of the language. Section 3.3 depicts the guarantees of the DSL. Section 3.4 presents its integration into MiniZinc and discusses the possible limitations of our approach. Finally, Section 4 evaluates the DSL on both parsing and solving stages and shows, on a use-case, how helpful prototyping propagation engines can be.

2 Constraint Programming Background

Constraint programming is based on relations between variables, which are stated by constraints. A *Constraint Satisfaction Problem* (CSP) is defined by a triplet $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ and consists of a set of variables $\mathcal{V}$, their associated domains $\mathcal{D}$, and a collection of constraints $\mathcal{C}$. The domain $D(v) \in \mathcal{D}$ associated with a variable $v \in \mathcal{V}$ defines a finite set of integer values v can be assigned to. An *assignment* of a variable v to a value x is the reduction of its domain to a singleton, $D(v) = \{x\}$. A constraint $c \in \mathcal{C}$ is defined over a set of variables $V(c) \subseteq \mathcal{V}$. It is equipped with a Boolean function which states wether an assignment of all its variables $V(c)$ is *valid* according to its semantic. A constraint c is said to be *satisfiable* if there exists a valid assignment of its variables. A solution to a CSP is an assignment of all variables of $\mathcal{V}$ such that all the constraints of $\mathcal{C}$ are simultaneously satisfied. A constraint is characterized by its behavior regarding modifications of the domains of variables. A constraint c is equipped with one or more filtering algorithms, called *propagators*. The set of propagators of a CSP is denoted $\mathcal{P}$. A propagator $p \in \mathcal{P}$ removes, from the domains $\mathcal{D}$, values that do not correspond any more to a valid assignment. Such a reasoning is achieved through a dedicated method, named $\texttt{revise}(p, v)$,[1] which filters the variables of the propagator p according to a modification of the variable v and returns the set of modified variables. Let $V(p)$ be the set of variables of p, and $P(v)$ be the set of propagators associated with a variable v, and $P(c)$ be the set of propagators of c.

Constraint programming is based on *constraint propagation* [10,18]. Propagators (associated with constraints) perform domain reductions on variables. Transmitting those domain reductions through the constraint network is necessary because they may affect the validity of constraints assignments. The transmission process is called *propagation* [1]: it consists in scheduling and executing the propagators that may deduce, from those reductions, new value removals. The process is iterated until no further modification is identified or a domain is wiped out. In order to be efficient, modern constraint solvers need to implement trendy features [16]. For this purpose, they have to deal explicitly or implicitly with a fine-grained information which, at least, has to consider each pair propagator-variable $\langle p, v \rangle$, $p \in \mathcal{P}$ and $v \in V(p)$. Such an association is called an *arc* [3] and will be denoted $a = \langle p, v \rangle$. We add the following notations to ease arc manipulation: $A(v)$ denotes the arcs connected to variable $v \in \mathcal{V}$; $A(p)$ denotes the arcs connected to propagator $p \in \mathcal{P}$; Given an arc $a = \langle p, v \rangle$, $V(a)$ denotes its variable v, $P(a)$ denotes its propagator p.

Algorithm 1 *Arc*-oriented propagation algorithm

1: Q initialized with all the arcs $A(v)$ connected to current decision variable v
2: **while** $Q \neq \emptyset$ **do**
3: $\quad a \leftarrow \texttt{remove}(Q)$ ▷ remove and retrieve an arc from Q
4: $\quad Q \leftarrow Q \cup \{A(w) \mid w \in \texttt{revise}(P(a), V(a))\}$ ▷ push all the arcs associated with the variables modified by propagator $P(a)$
5: **end while**

[1] On the one hand, event-based arc-oriented engines and variable-oriented engines input the modified variable to the propagator. On the other hand, event-based propagator-oriented engines need to explicitly compute (or to explicitly maintain) the modified variables.

Algorithm 1 depicts the generic sketch of a constraint propagation procedure when reasoning with an arc representation. The propagation algorithm is triggered by a domain reduction, for instance when a decision is applied (line 1). Next, until there is no arc to revise in the data structure Q, an arc a is removed from Q (line 3) and, all the arcs associated with the variables modified by the filtering algorithm of the $P(a)$ propagator are added to Q (line 4). The revision of the propagator $P(a)$ may detect a *failure*, for instance a domain becomes empty, and make the algorithm stop. Implementation details are not provided here for sake of simplicity.

The way Q is *oriented* may have an influence on the way propagators will be implemented. The orientation is stated by the kind of items Q handles. Alternatives to arcs are to schedule variables [2] (Algorithm 2) or, to schedule propagators [5] (Algorithm 3). In the case of a *variable-oriented* propagation algorithm, when a variable is removed from Q (line 3), its propagators have to be executed sequentially and the modified variables are added to Q (line 4). In the case of a *propagator-oriented* propagation algorithm, when a propagator is removed from Q (line 3), it is executed and all the propagators for which at least one variable has been modified are added to Q (line 4). The worst case complexity remains unchanged in all cases. This means the worst total number of operations applied is equivalent, even if the maximum number of items added into Q may vary, and the execution order as well. In Algorithm 1, Q may contain at most $\mathcal{O}(n \times m)$ arcs, while only $\mathcal{O}(n)$ (resp. $\mathcal{O}(m)$) may be stored by Algorithm 2 (resp. Algorithm 3). Indeed, Algorithms 2 and 3 require to respectively compute the propagator to apply ($\texttt{revise}(P(a), v)$, line 4 of Algorithm 2) and the modified variable ($\texttt{revise}(p, V(a))$, line 4 of Algorithm 3).

Algorithm 2 *Variable*-oriented propagation algorithm

1: Q initialized with the current decision variable v
2: **while** $Q \neq \emptyset$ **do**
3: $\quad v \leftarrow \texttt{remove}(Q)$
4: $\quad Q \leftarrow Q \cup \{\bigcup_{a \in A(v)} \texttt{revise}(P(a), v)\}$
5: **end while**

Algorithm 3 *Propagator*-oriented propagation algorithm

1: Q initialized with all the propagators $P(v)$ connected to current decision variable v
2: **while** $Q \neq \emptyset$ **do**
3: $\quad p \leftarrow \texttt{remove}(Q)$
4: $\quad Q \leftarrow Q \cup \{P(w) \mid w \in \bigcup_{a \in A(p)} \texttt{revise}(p, V(a))\}$
5: **end while**

Global constraints in constraint solvers have brought up the importance of carefully organizing execution of propagators with respect to their complexity. Thus, it has been shown that it is worth scheduling propagators of global constraints carefully. On the one hand, in variable-oriented solvers, this is achieved for instance, by adding a queue dedicated to *coarse* propagation [6,11]. Such a mechanism can be improved, for instance by applying *watched literals* [12] to propagators. On the

other hand, [16] introduced the notion of *staged propagators* (combining several distinct propagators into a single one with an internal state variable) and proposed a range of priorities to discriminate propagators, in propagator-oriented solvers. [14] introduced *advisors* as an additional concept. Obviously, these approaches are very similar: they aim at reducing the number of scheduling calls (*i.e.*, for Algorithms 1 to 3, the number of times an item is added in Q) and increasing filtering efficiency.

In addition to those works on scheduling conditions, the *revision ordering* of elements to propagate has also been studied. The revision ordering defines how an element is selected and removed from the set Q. It is generally acknowledged that a queue, which chooses the oldest item first, is a good choice for the set Q as it guarantees fairness. However, alternatives have been proposed. On the one side, [9] introduced dynamic element selection based on a criterion (*e.g.*, domain size of variable or propagator arity). On the other side, [16] proposed a priority bucket queue, that is, a seven-level priority queue. A propagator is then scheduled into the most relevant queue with respect to its dynamic priority.[2] Recent progress in reducing propagation overhead has been achieved introducing propagator groups [17]. Nevertheless, it must be noted that the implementation of groups is left to the user.

Finally, unlike search strategies that are commonly available in the forms of portfolios or combinators [21], propagation engines are more likely imposed on users in constraint solvers. Such a statement depends much on the fact that defining a propagation algorithm still remains a difficult task. In addition to global knowledge of a solver architecture, it takes an expert programmer in order not to deteriorate the performance and correctness of the solver. For these reasons, we introduce a Domain Specific Language (DSL) to prototype a propagation engine algorithm, in a fast and robust fashion.

3 A DSL to describe Propagation Engines

In order to provide a convenient expression, or declaration, of a propagation engine by a user, we introduce a *Domain Specific Language* [7]. First, we give a list of solver requirements to fully benefit from the DSL. Second, we introduce the DSL in two steps: (1) declaration of groups of arcs and (2) structure and combination of these groups. Then, we list the properties and guarantees of our DSL, and sketch how it can be implemented. Finally, we briefly present MiniZinc [20] and we describe its extension to support our DSL.

3.1 Requirements

A few characteristics are required to fully benefit from the DSL. Presumably, due to their positive impact on efficiency, modern constraint solvers already implement these techniques:

[2] Dynamic priority combines arity and priority to provide an accurate evaluation of a propagator execution cost [16].

Fig. 1: The DSL entry point.

⟨*propagation_engine*⟩ ::= ⟨*group_decl*⟩+ ⟨*structure_decl*⟩ ;

- *Staged* propagators [16]: propagators are discriminated thanks to their priority. The priority should determine which propagator to run next: lighter propagators (in the complexity sense) are executed before heavier ones.
- Grouped propagators [17]: a *controller propagator* is attached to each group of propagators;
- Open access to variable and propagator properties: for instance variable cardinality, propagator arity or propagator priority.

Note that properties mentioned in the latter point may be evaluated dynamically during propagation. Obviously, dynamic evaluation comes at a cost and may require specific data structures [9].

To be more flexible and more accurate, we assume that all arcs from $\mathcal{A}$, the set of arcs of a CSP, are explicitly accessible. This is achieved by explicitly representing all of them and associating them with watched literals [12] or advisors [14]. Explicit representation of all the arcs comes with a memory overhead: $\mathcal{O}(|\mathcal{V}| \times |\mathcal{P}|)$ additional objects are required, each of them maintaining two pointers (to a variable and a propagator). On the other hand, arcs bring flexibility by providing access to properties of both their variable and their propagator. Moreover, they may be organized by variables, propagators, or properties (for instance, propagators priority).

3.2 Domain Specific Language

A program in the DSL defines a global ordering over the set of arcs $\mathcal{A}$. The DSL is divided in two parts (Figure 1): first, the group of arcs definition, second, propagation engine structure description based upon these groups. This two-step process decouples groups and their combination. We discuss these two parts individually in the following.

The syntax is presented in standard BNF adopting the following conventions: typewriter `tt` indicates a terminal; italic ⟨*it*⟩ indicates a non-terminal; brackets [*e*] indicate *e* optional; double brackets [[a-z]] indicate a character from the given range; the Kleene plus *e*+ indicates a sequence of one or more repetitions of *e*; the Kleene star *e*⋆ indicates a sequence of zero or more repetitions of *e*; ellipsis *e*,... indicates a non empty comma-separated sequence of *e*; alternation *a*|*b* indicates alternatives.

3.2.1 Group assignment declaration

We now present the part of the DSL dedicated to group declaration. The notion of group is defined in [17]. The aim of groups is simple but powerful: controlling the execution of groups of propagators by deferring scheduling and execution of these propagators to user-supplied routines. Here, we replace propagators by arcs. Groups of arcs are defined with assignment rules (Figure 2). The user only knows variables or constraints at the modeling stage. Consequently, variables and constraints and their respective properties can be freely referenced in this part of the

Fig. 2: Group definitions

⟨*group_decl*⟩ ::= ⟨*id*⟩ `:` ⟨*predicates*⟩`;`
⟨*id*⟩ ::= [[a-zA-Z]][[a-zA-Z0-9]]⋆
⟨*predicates*⟩ ::= ⟨*predicate*⟩ | `true`
| `(`⟨*predicates*⟩ `((&&` | `||)` ⟨*predicates*⟩`)+)`
| `!`⟨*predicates*⟩
⟨*predicate*⟩ ::= `in(`⟨*var_id*⟩ | ⟨*cstr_id*⟩`)`
| ⟨*attribute*⟩ ⟨*op*⟩ (⟨*int_const*⟩ — `"string"`)
⟨*op*⟩ ::= `==` | `!=` | `>` | `>=` | `<` | `<=`
⟨*attribute*⟩ ::= `var`[`.(name`|`card)`]
| `cstr`[`.(name`|`arity)`]
| `prop`[`.(arity`|`priority`|`prioDyn)`]
⟨*int_const*⟩ ::= [+−][[0-9]][[0-9]]⋆
⟨*var_id*⟩ ::= `-`⋆⟨*id*⟩
⟨*cstr_id*⟩ ::= `-`⋆⟨*id*⟩

DSL. However, propagators are not modeling objects, so they can be referenced only through their properties. We focus below on properties available in many constraint solvers.

A group is declared by an identifier and a list of predicates. The identifier ⟨*id*⟩ is a unique name starting with a letter, for instance `G1`. A predicate is a Boolean-valued function $\mathcal{A} \rightarrow \{\texttt{true}, \texttt{false}\}$ and defines a group membership. A predicate is *in extension* if its declaration is based on a specific variable or constraint (pointed out with `var_id` and `cstr_id`). Every arc associated with the variable or the constraint is a candidate for the group. For instance, `in(`v_1`)` defines a set of arcs A such that for all $a \in A$, $V(a) = v_1$; `in(`c_1`)` defines a set of arcs B such that for all $a \in B$, $P(a) \in P_{c_1}$. A predicate is said to be *in intension* if its declaration is based on properties. Any arc of the resulting set satisfies the properties. Intension predicates are defined by an attribute ⟨*attribute*⟩, an operator ⟨*op*⟩ and an integer value or a string expression. An ⟨*attribute*⟩ refers to a property of a variable (`var.name`, the name of the variable; `var.card`, its cardinality), a constraint (`cstr.name`, the name of the constraint; `cstr.arity`, its arity) or a propagator (`prop.arity`, the arity of the propagator; `prop.priority`, its static priority; `prop.prioDyn`, its dynamic priority). For instance, `prop.priority` < 4 defines a set of arcs C such that the priority of all propagators involved in arcs of C is strictly less than 4. Predicates can be composed with logical operators `&&`, `||` and `!`.

The ⟨*group_decl*⟩ may be adapted to each solver implementation by extending ⟨*attribute*⟩, that is, adding new properties of constraints and variables. ⟨*group_decl*⟩ can be model-dependent, by naming variables and constraints of a specific problem, or totally independent by filtering only on properties. The aim of ⟨*group_decl*⟩ is to define sets of arcs. This part of the DSL is evaluated in an imperative way: group definitions are evaluated top-down and left-to-right. Indeed, several predicates can return `true` for the same arc, but an arc can only belong to a single group. So, once an arc belongs to a group, it is not a candidate anymore for other groups defined later.

3.2.2 Structure declaration

The second part of the DSL (Figure 3) covers the data structure definition based upon groups of arcs. The objective here is twofold: “where” to store an arc and

Fig. 3: Data structure of the propagation engine

⟨*structure*⟩	::=	⟨*struct_ext*⟩ | ⟨*struct_int*⟩
⟨*struct_ext*⟩	::=	⟨*coll*⟩ `of` {⟨*elt*⟩`,`...} [`key` ⟨*comb_attr*⟩]
⟨*struct_int*⟩	::=	⟨*id*⟩ `as` ⟨*coll*⟩ `of` {⟨*many*⟩} [`key` ⟨*comb_attr*⟩]
⟨*elt*⟩	::=	⟨*structure*⟩
	|	⟨*id*⟩ [`key` ⟨*attribute*⟩]
⟨*many*⟩	::=	`each` ⟨*attribute*⟩ `as` ⟨*coll*⟩ [`of` {⟨*many*⟩}] [`key` ⟨*comb_attr*⟩]
⟨*coll*⟩	::=	`queue(`⟨*qiter*⟩`)`
	|	[`rev`] `list(`⟨*liter*⟩`)`
	|	[`max`] `heap(`⟨*qiter*⟩`)`
⟨*qiter*⟩	::=	`one` | `wone`
⟨*liter*⟩	::=	⟨*qiter*⟩ | `for` | `wfor`
⟨*comb_attr*⟩	::=	(⟨*attr_op*⟩`.`)⋆⟨*ext_attr*⟩
⟨*ext_attr*⟩	::=	⟨*attribute*⟩ | `size`
⟨*attr_op*⟩	::=	`any` | `min` | `max` | `sum`

"how" to select an arc. The structure declaration results in building a hierarchical tree structure based on a collection ⟨*coll*⟩.

A ⟨*coll*⟩ is implemented by an abstract data type (ADT) and is a collection of other ADT and arcs. Thus, there is no guarantee that a ⟨*coll*⟩ exclusively handles arcs. An ADT defines how its items will be scheduled and removed for revision. There are three of them: `queue`, `list` and `heap`. Each of them comes with an iterator which describes the traversing strategy. A `queue` is a *first-in-first-out* collection of items. A `list` is a statically ordered collection of items (`rev` reverses the list). A `heap` is a dynamically ordered collection of items according to a criterion (the default comparison function is $\leq$, the keyword `max` changes it to $\geq$).

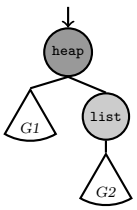
Basic iterators `one` and `wone` are defined for any ⟨*coll*⟩. `one` picks and removes one element from a ⟨*coll*⟩. `wone`, short version of 'while one', calls `one` until the ⟨*coll*⟩ becomes empty. A `list` comes with two extra iterators: `for` and `wfor`. `for` sequentially picks and removes elements in the list in increasing order of an index. `for` does not allow going backwards into the list nor interrupting the traversing. The ⟨*coll*⟩s are mutable; New items may be scheduled in a ⟨*coll*⟩ during an iteration. Since the `for` iterator does not allow going backward, there is no guarantee that the ⟨*coll*⟩ is empty at the end of the iteration. Thus, we introduce `wfor`, short version of 'while for', which calls `for` until the ⟨*coll*⟩ becomes empty.

A `heap` requires the declaration of a comparison criterion (which is optional for `list`). The criterion must be static for a `list` and dynamic for a `heap`. As said earlier, there is no guarantee that a ⟨*coll*⟩ exclusively handles arcs. The sorting criterion should be defined by (a) a `key` and an ⟨*attribute*⟩ for an arc or (b) a `key` and a ⟨*comb_attr*⟩ for a set of items. The evaluation of a set is done by evaluating arcs of the set (combining ⟨*attr_op*⟩). An extension of ⟨*attribute*⟩, named ⟨*ext_attr*⟩, is introduced in Figure 3. An ⟨*ext_attr*⟩ allows to reach any item's attribute in a collection of items, such as its `size`. For instance, Figure 4 depicts a heap-based propagation engine declaration. The top-level heap requires its composed items to be comparable. All the arcs of `G1` are evaluated by returning the cardinality of their variable (Line 2). The second group is a list of arcs of `G2`. The evaluation of a list is not possible and it should be delegated to one of its component arcs. Thus, an arc of `G2` is selected arbitrarily (`any`) to get the cardinality of its variable (Line 3). The keyword `any` indicates both to delegate the criteria evaluation to an item of `list` and to select it arbitrarily.

Fig. 4: Heap-based propagation engine.

```
1. heap(wone) of {
2.     G1 key var.card,
3.     list(for) of {G2} key any.var.card
4. }
```

(a) Declaration.

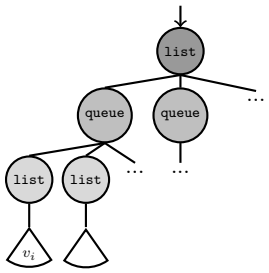


(b) Tree representation

Fig. 5: Example of regular structure.

```
1. G as list(wfor) of{
2.   each prop.priority as queue(for) of{
3.     each var as list(for)}
4. }
```

(a) Declaration.



(b) Tree representation.

Now we describe how to use abstract data structures in a complete structure. The entry point of a structure declaration is ⟨*structure*⟩. A ⟨*structure*⟩ defines the top-level structure which pilots the propagation. A ⟨*structure*⟩ can be defined either in extension (⟨*struct_ext*⟩) or in intension (⟨*struct_int*⟩).

A ⟨*struct_ext*⟩ defines a collection ⟨*coll*⟩ which is composed of a list of elements ⟨*elt*⟩. An ⟨*elt*⟩ can either reference a group identifier and an ordering instruction (⟨*id*⟩ [`key` ⟨*attribute*⟩]) or another ⟨*structure*⟩. Figure 4 depicts a `heap` that handles arcs of `G1` and a list.

A ⟨*struct_int*⟩ defines a regular structure based on a single quantifier `each-as`. A ⟨*struct_int*⟩ allows to express, in a compact way, nested structures, whereas ⟨*struct_ext*⟩ would require a more verbose expression and more likely introduce errors. ⟨*struct_int*⟩ expresses that the arcs of a group must be spread into collections under specific conditions, given by an attribute. It takes a set of arcs defined by a group identifier as input and *generates* as many ⟨*coll*⟩ as required by the ⟨*many*⟩ instruction, possibly nested. ⟨*many*⟩ induces a loop over ⟨*attribute*⟩ values and groups items with the same ⟨*attribute*⟩ value together in the same ⟨*coll*⟩. Figure 5 shows a nested regular structure declaration. The construction of the data structure takes `G` as input: as many sets of arcs as `prop.priority` among arcs of `G` are created (Line 2). Remind that [16] defines 7 priorities. Then, all sets are considered one by one and their arcs are organized by variables in lists. All lists are added into the `queue` which corresponds to the desired `priority`. The queues are put together in the top-level list. The assignment to the right collection may be achieved dynamically during the resolution process when using a dynamic criterion.

3.3 Properties and Guarantees of the DSL

We now describe all the properties and guarantees supported by our DSL proposal.

Solver independent description. Our DSL does not rely on specific solver requirements (beyond those listed in Section 3.1) and it can be implemented on top of a range of different constraint solvers. When only a part of these requirements is supported by the solver, the DSL is restricted accordingly. First of all, we assume that solvers provide full access to variable and propagator properties because these are at the heart of most of the criteria used to schedule propagation. If a solver does not support the arc representation, the expressivity of the DSL would be limited to the solver orientation (variable or propagator, cf. Section 2) and group declarations. In other words, the DSL expresses propagation policies according to several supported propagation engine orientations. If a solver represents arcs but does not support group declaration then the expressivity of the DSL is reduced to the revision order of the unique top-level set of arcs. Variable-oriented solvers are presumably better prepared for arc representation, as theirs propagators are informed of the exact modified variable. Propagator-oriented solvers, on the other hand, recover the modified variables, thanks to advisors for example.[3]

Expressivity. Our DSL covers commonly used data structures and characteristics (from simple queue of items to seven-bucket queues). Moreover, it goes beyond them by combining them together and with access to solver properties. This enables the description of a great variety of propagation engines in a concise way, including mixing orientations.

Extensibility. The DSL can be extended in two ways. First, new attributes can be introduced to make group definition more concise. Second, new collections and new iterators can provide new propagation schemes. In particular, this gives the opportunity to declare incomplete propagation, such as those described in [13].

Ensured coverage. All the arcs of a model are represented within the propagation engine. The expressivity of our DSL allows descriptions of incomplete propagation engines (in which one or more arcs may be absent). Thus, the resulting propagation engine may ignore a subset of $\mathcal{A}$, and consequently, a subpart of the model. This lack of coverage can be detected, and the user warned or the execution stopped. Alternatively, a last generic group can collect arcs not assigned to any group to ensure coverage.

[3] A solver core refactoring is required to fulfill those preliminary conditions and fully benefit from the DSL expressivity. Such a refactoring is a very critical task for developers. We believe event-based constraint solvers are more prone to supporting arc-oriented filtering algorithm. In addition, such a refactoring not only brings more flexibility into constraint solvers but opens new perspectives in the resolution process. Naturally, this does not question native engines (*i.e.*, not DSL-driven engines), but it comes in addition to these ones. It may simplify the design of propagator groups as well.

Unique propagation. One arc may satisfy multiple predicates and be eligible to different groups. The top-bottom left-right evaluation of the DSL ensures that each arc is only represented once in the propagation engine. Thus, an event occurring on a variable cannot be treated more than once. This ensures that our DSL does not change the complexity of the standard propagation phase.

Conformance. Arcs related to no pair (p,v) of the model are not represented within the propagation engine. Thanks to this property, no *senseless* arc is scheduled and propagated during the propagation process. This preserves the time complexity of the propagation engine.

Completeness. Completeness is ensured by a top-level collection with a "while" condition, such as `wone` and `wfor`. Whatever the substructure iterators are, the top-level group ensures that there is no pending event before terminating its execution. Selecting `one` or `for` may remove the completeness guarantee. However, [13] precisely explains how incomplete propagation could be useful.

Reliability. Our DSL makes it possible to describe ill-formed, yet still syntactically correct, propagation engines. For instance: a group may be empty, a group definition may be unused in the structure declaration, a structure may nest $\langle many \rangle$ based on the same $\langle attribute \rangle$, a structure may contain no arc or only one arc, etc. However, we check this and the user is warned against such rule violations. Further work could be to automatically apply the corresponding simplifications (suppress empty groups, flatten unnecessary nested definitions, etc.).

Complexity. Both "Conformance" and "Unique propagation" properties ensure to maintain the complexity of the propagation engine algorithm. However, the declaration of ill-formed propagation engines or excessive usage of dynamic criteria may have a non negligible impact on the global performance of a propagation engine. Users should be aware of such degradation induced by the generalization of the architecture during the prototyping of the propagation engine and its evaluation. But, nevertheless the DSL prevents from creating engines which do not end, and thus usable for the purpose of prototyping.

3.4 Implementation

We now present an implementation of the DSL. Extending a modeling language is a natural way to implement our DSL: it provides access to model objects, such as variables and constraints, but maintains solver independence. We selected MiniZinc [20] to be extended with our DSL. It is a simple yet expressive CP modeling language. It is suitable for modeling problems for a range of solvers. Then, we selected ANTLR[4] to interpret the grammar. It enables a clear separation between the lexical analyzer (the *lexer*), the recognition of sentence structure (the *parser*) and the translation into solver instructions (the *tree parser*). That simplifies the reusability of the lexer and the parser, only the tree parser needs to be adapted to solvers.

[4] http://www.antlr.org/

Fig. 6: MiniZinc model of the `magic sequence` extended with our DSL .

```
1:  include "globals.mzn";
2:  annotation name(string: s);
3:  annotation engine(string: s);
4:  int: n;
5:  array [0..n - 1] of var 0..n: x;
6:  constraint count(x,0,x[0])::name("c0");
7:  constraint forall (i in 1..n - 1) (count(x, i, x[i]));
8:  solve
9:  :: engine("
10: G1: in(\"c0\");
11: All: true;
12: list(wone) of {queue(wone) of{G1},
13:       All as queue(wone) of {each cstr as list(wfor)}};
14: ")
15: satisfy;
```

In practice, we need to add the propagation engine description inside the MiniZinc model. This is made with two kinds of annotations: the first one to add the description of the propagation engine, the second one to point out a constraint by naming it. Even though MiniZinc enables the definition of complex annotations, such as the ones used to declare search strategies, the annotations presented here are basic, *i.e.*, based on strings. This protects against reliance on a specific modeling language. Figure 6 shows an example of a MiniZinc model, where the propagation engine declaration has been added. Lines 1, 4-8 and 15 detail a classical model for the `magic sequence` problem (prob019, [19]) of size 5. The required annotations for naming constraints (line 2) and describing the engine (line 3) must be declared in the header of the model. The first constraint (line 6) is annotated by `::name("c0")` to name it. Lines 9-14 describe the propagation engine behavior: two groups are defined. The first group `G1` (Line 10) stores arcs involved in the constraint named `"c0"`; the second group `All` (Line 11) retains remaining arcs, thanks to the top-bottom left-right evaluation. Then, the structure of the propagation engine is defined by the top-level list composed of two queues (Line 12-13). The first queue stores arcs of group `G1` (Line 12); the second one is composed of lists of arcs, organized by constraints (Line 13). This propagation engine forces to always treat arcs occurring on the constraint `"c0"` before treating other arcs.

MiniZinc comes with a low-level solver-input language called FlatZinc. FlatZinc is the target language which MiniZinc models are translated into.[5] However, the MiniZinc to FlatZinc translation may induce a reformulation of defined constraints based upon primitive constraints. To focus on the DSL and its applicability, only built-in constraints and overloaded global constraints (that is, supported by the solver) can be used. Global constraints that rely on predicate definitions (See [20] for more details) are excluded: even if an original declaration could be transformed the same way constraints are decomposed, there is no guarantee that the resulting declaration fulfills the original intention. Indeed, consider a description asking for propagating as late as possible an `AllDifferent` constraint that ultimately through reformulation might be transformed into a clique of differences: this would be counterproductive and misleading. This is mainly due to predicates

[5] Annotations for describing the engine and naming constraint are transmitted to the FlatZinc model when the MiniZinc model is flattened.

and attributes related to group definition our DSL imposes. For instance, in the case of a DSL-driven engine based on priority, organizing arcs by priority may have dissimilar behavior: either it includes a global constraint, with potentially high priority, or its reformulation is based on a set of built-in constraints, with potentially low, and distinct, priorities. A recommended approach could be that MiniZinc supports our DSL, and provides, together with global constraint reformulation, a default structure declaration for each reformulation, based on the built-in constraints. In addition, the impact of a reformulation on the efficiency of propagation has not yet been addressed formally, and by itself represents an interesting research subject. We believe the introduction of the DSL is a good starting point for modelers and developers to prototype specific propagation engines.

Algorithm 4 presents the operational semantics of the propagation engine declaration in Figure 6, lines 9-14. It shows interpretations of the main steps of the DSL parsing and, the main steps of a resolution based on a DSL-driven engine. The main program, not detailed here, calls the two first procedures (Group assignment declaration and Structure declaration) during the parsing phase and the two last procedures (Schedule and Propagate) during the resolution phase. First, the group assignment declaration (lines 1-10) consists in an iteration over the arcs of the model to assign each of them to a group. The assignment predicate of $G1$ is evaluated first because it is declared first. Second, the structure declaration (lines 11-22) consists in a sequence of structure construction and initialization, from the bottom-level structures ($q1$ and ls) to the top-level one ($l1$). Note that arcs of the group $G1$ are organized by constraints in an intermediary structure (line 17). Algorithm 4 also suggests how to interpret the two main functions a propagation engine comes with: how to schedule an arc (lines 23-34) and how to select an arc for propagation (lines 35-50). Nested structures imply that scheduling an arc forces to schedule any group between the arc and the top-level group. When an arc has to be selected for revision, the explicit or implicit order between groups depicted during the declaration is preserved, iterators behave similarly. For instance, the arcs of $G1$ must be treated before any other arcs (lines 37-40). When $q1$ is empty then, $q2$ is empty too. The `wfor` iterator declared in Figure 6 line 13, is transformed into a combination of a while-loop and a for-loop (lines 43-47).

Basically, groups define a hierarchy of nested partitions of the arcs, and each propagation policy is implemented by a specific container iterator. This way, the operational semantics of any propagation engine defined in the DSL corresponds to a tree of containers and their corresponding nested iterators. Our implementation is based on this tree of containers and their iteration methods.

4 Experimental Evaluation

Most of the time, the data structures used to build a propagation engine have very low complexities. For instance, a circular queue or a bitset provides `add()` and `remove()` operations in constant time, a binary heap provides `add()`, `extractMin()` and `update()` operations in $\mathcal{O}(\log n)$ time, where n is the size of the heap. Even though these operations are critical, the efficiency of the resolution phase is more linked to the order in which propagators are executed. There can be an overhead in parsing the DSL, due to both the data loading and guarantee checks. But, the

Algorithm 4 Evaluation scheme of propagation engine described in Figure 6

```
Require: Arcs: set of arcs of the model
 1: procedure GROUP ASSIGNMENT DECLARATION                    ▷ Figure 6, lines 10-11
 2:     Declare G1, All as set of arcs
 3:     for each arc a in Arcs do
 4:         if a is associated with a propagator of c0 then          ▷ Predicate of G1
 5:             move a to G1
 6:         else if true then          ▷ Predicate of All, i.e., remaining arcs
 7:             move a to All
 8:         end if
 9:     end for
10: end procedure
```

```
Require: G1, All: set of arcs
11: procedure STRUCTURE DECLARATION                           ▷ Figure 6, lines 12-14
12:     Declare q1 as queue
13:     Fill q1 with arcs of G1
14:     Declare q2 as queue
15:     Declare ls as array of lists
16:     Declare m as map
17:     Map arcs of All to constraints in m
18:     Fill ls with sets of arcs defined by m
19:     Fill q2 with lists of ls
20:     Declare l1 as list
21:     Fill l1 with queues q1, q2
22: end procedure
```

```
Require: a: an arc
23: procedure SCHEDULE                          ▷ Operations to execute on arc scheduling
24:     Declare tmp as schedulable object
25:     tmp ← a
26:     repeat
27:         if tmp is not present in its group then          ▷ if tmp is not already scheduled
28:             Add tmp to its group          ▷ the selection of the group may be dynamic
29:             tmp ← group of tmp
30:         else
31:             return          ▷ tmp is already scheduled, so do the higher level groups
32:         end if
33:     until c is the top level group                ▷ The top-level group belongs to itself
34: end procedure
```

```
35: procedure PROPAGATE                          ▷ Operations to execute on a propagation call
36:     while l1 is not empty do
37:         while q1 is not empty do
38:             Pop the first arc a of q1 and execute it          ▷ i.e., revise(P(a), V(a))
39:             Execute P(a) with arcs involving V(a)
40:         end while
41:         while q2 is not empty do
42:             Pop a list l of q2
43:             while l is not empty do                                  ▷ while-loop and ...
44:                 for i in 1...size(l) do          ▷ ... for-loop to ensure "wfor" behavior
45:                     Remove and execute the i^th arc a of l, if present   ▷ i.e., revise(P(a), V(a))
46:                 end for
47:             end while
48:         end while
49:     end while
50: end procedure
```

overhead of using an interpreted propagation engine should be limited to indirections induced by group composition and criteria evaluations.

All the following experiments were done on a Macbook Pro with a 6-core Intel Xeon at 2.93Ghz running on MacOS 10.6.8, and Java 1.6.0_35. The empirical results are based on 20 runs (the standard deviation is always less than 5%) and the tables report parsing times (time spent parsing the MiniZinc file) in millisec-

onds and solving times in seconds. The evaluation of our DSL and its interpretation have been implemented in Java for Choco [6]. Choco is a Java library for constraint programming. The source code has been refactored to support the prerequisites listed in Section 3.1. Furthermore, three native (*i.e.*, not DSL-driven) propagation engines are implemented in the library: a propagator-oriented one `prop`, a variable-oriented one `var` and a seven-queue dynamic priority one `7qd` [16]. The first experimental evaluation is made on a basis of 39 examples extracted from 21 problems[6] of the MiniZinc [20] distribution. More details on these benchmarks are available on demand. They give both an evaluation of the parsing phase and the position of interpreted propagation engines among native propagation ones. Then, we present a small use-case, based on the `Golomb ruler` problem. It presents how prototyping propagation engines can be easily achieved using the DSL.

4.1 Engine descriptions

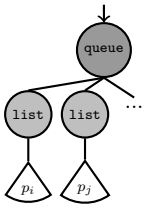
This section gives a simulation of three state-of-the-art native engines (`prop`, `var` and `7qd`) with the help of the DSL, and then compares the treatment of MiniZinc files with and without engine declaration. For each native engine, its DSL-driven version is guaranteed to build the same tree search and to preserve the order and the number of propagator executions. This enables us to exemplify the expressivity of our DSL.

First we present how to declare `prop`, a propagator-oriented propagation engine, with the DSL (Figure 7). The structure takes the overall list of arcs of the problem as input. Then, arcs are organized by propagators in lists. Each list defines an incomplete iteration (`for`) but the master queue ensures completeness of the propagation (`wone`). Second, we present a declaration of `var`, a variable-ori-

Fig. 7: A propagator-oriented propagation engine.

```
1: All:true;
2: All as queue(wone) of {
3:     each prop as list(for)
4: }
```

(a) Declaration.



(b) Tree representation.

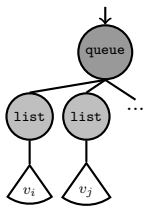
ented propagation engine, with the DSL (Figure 8). It looks like the previous one, but arcs are organized by variables. Finally, we present how to declare `7qd` with our DSL, a seven-queue priority dynamic propagator-oriented engine (Figure 9). As the previous ones, it takes all arcs of the problem as input. These arcs are distributed into queues with respect to the dynamic priority of their propagators.

[6] The selected problems are: `bacp`, `bibd`, `black-hole`, `CostasArrays`, `debruijn-binary`, `fastfood`, `fillomino`, `ghoulomb`, `golomb`, `jobshop`, `langford`, `latin-squares`, `magicseq`, `market_split`, `nonogram`, `p1f`, `pentominoes`, `radiation`, `rcpsp`, `slow_convergence`, `still_life`.

Fig. 8: A variable-oriented propagation engine.

```
1: All:true;
2: All as queue(wone) of {
3:     each var as list(for)
4: }
```

(a) Declaration.



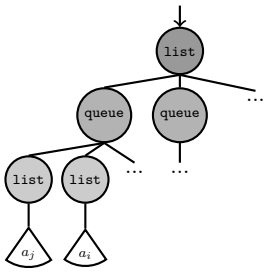
(b) Tree representation.

For each available priority, arcs are organized by propagators in lists. As the evaluation of the priority is dynamic, the selection of the queue to put a list in is made through the evaluation of any arc's propagator within a list. A top-level list ensures both completeness of the propagation (`wone`) and treatment of arcs associated with higher priority propagators before lower priority ones. Note that the `any` keyword is doubled. This is necessary to delegate the evaluation of the `priority` of a queue to an assessable item, any arc among its lists. The first `any` points to an item of a queue, *i.e.*, a list, the second `any` points to an item of the list, *i.e.*, an arc. This arc is the representative of the queue, and its propagator is used for the evaluation of the queue.

Fig. 9: A priority-oriented propagation engine.

```
1: All: true;
2: All as list(wone) of {
3:     each prop.prioDyn as queue(one) of {
4:         each prop as list(for)
5:     }key any.any.prop.prioDyn
6: }
```

(a) Declaration.



(b) Tree representation.

4.2 Parsing and Solving evaluations

The aim of the prototyping process is to declare and to evaluate rapidly, yet safely, propagation engines. On the one hand, the parsing phase is not critical in a prototyping approach, but it should not penalize the testing process too much. On the other hand, the resolution step is critical. Interpreted versions of engines cannot be competitive, in terms of efficiency, with native implementations; However, we show that the overhead is small enough to make them useful for prototyping, but especially that the runtime order among the interpreted engines is generally the same as that of native ones. The cost of using an interpreted engine must be minimal to validate our approach as a relevant prototype tool. Table 1 reports the overhead

of parsing MiniZinc files for the DSL interpreted propagation engine compared to their native version. The parsing operation includes file loading, lexical analysis, sentence structure recognition and the translation into solver instructions. Table 1 also reports overhead of solving examples with interpreted propagation engines compared to their native versions. The runtime excludes the parsing time. This step is critical, and the cost of using such an approach must be small enough to validate it as a good prototype tool.

Table 1: Average overheads induced by using the DSL, over 39 examples.

Engine	Parsing		Solving	
	avg.	std. dev.	avg.	std. dev.
`prop`	28.56%	19.34	9.33%	7.29
`var`	28.75%	20.15	7.01%	5.29
`7qd`	28.07%	17.64	13.20%	7.63

As expected, every single operation of the parsing is impacted by the addition of propagation engine declarations, and globally the parsing phase requires about 28.46% more time, on average (arithmetic mean of the runtime overhead). The details show that there is a direct relation between the parsing time overhead and the number of arcs: the more arcs there are, the longer the parsing takes, for instance this is the case for the problems `slow_convergence`, `non_non_fast`, `latin-squares`, `debruijn_binary` and `bibd` for which the parsing time requires at least 50% overhead. However, even if the overhead seems to be important, the parsing time is negligible compared to the solving phase (less than 200ms, excepted for `slow_convergence`). Moreover, this overhead comes not only from the necessity of arc representation in memory and property checking but it also depends on how arcs are organized. For instance, in examples where the number of variables is small compared to the number of propagators, parsing the DSL version of `var` is faster than the other ones. Finally, we observe that parsing `prop` and `7qd` is comparable. Organizing arcs by propagators is a common operation for the two declarations.

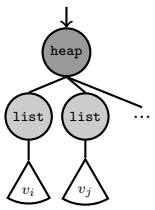
Concerning the solving phase, we can observe that the global average overhead of using an interpreted propagation engine is about 9.81% (arithmetic mean of the runtime overhead). Such overheads presumably come from lack of optimization interpreted versions suffer from. In an interpreted version, when an arc is scheduled for propagation, the list it belongs to should also be scheduled, if it was not already. In the native version, the second operation is natively supported. In a programming language that support direct optimization (such as C), the overheads may be more easy to reduce.

`var` and `prop` have smaller overheads than `7qd`. But, the DSL introduces two sources of potential weaknesses: multiple criteria evaluation and nested data structures. This is particularly the case for `7qd` which comes at a larger cost. In the native version, the evaluation of the `prioDyn` is done efficiently by requesting directly the propagator. In the interpreted version, each arc is evaluated on scheduling. Consequently, even though the native version of `7qd` has been shown to be very competitive, its interpreted version is penalized. As expected, basing engines on the DSL comes at a cost. But, it might be noticed that for 38 out of 39 (97.4%) examples, the runtime order among the interpreted engines is the same as the

Fig. 10: Heap-based propagation engine.

```
1: All: true;
2: All as heap(wone) of {
3:     each var as list(for) key any.var.card
4: };
```

(a) Declaration.



(b) Tree representation

runtime order among native engines. Moreover native versions of such propagation engines are very hard to implement efficiently. Although the DSL introduces memory and time overheads, it preserves guarantees and comes with an acceptable cost. Our next step is to study the contribution of the DSL on a use case and highlight how easy testing various propagation schemas can be.

4.3 `Golomb ruler`: a use case

We now present, on a use case, how easy it is to declare propagation engine prototypes. The process is the following: starting from common propagation engines, we declare state-of-the-art propagation engines, and finally we try out two additional strategies using our DSL.

The `Golomb ruler` problem (prob006, [19]) is defined "as a set of m integers $0 = a_1 < a_2 < \cdots < a_m$ such that the $\frac{m(m-1)}{2}$ differences $a_j - a_i, 1 \leq i < j \leq m$ are distinct. Such a ruler is said to contain m marks and is of length a_m. The objective is to find optimal (minimum length) or near optimal rulers". [9] showed that propagating through the selection of the smallest cardinality variable is a good strategy to improve the resolution of the `Golomb ruler`. We instrumented the MiniZinc model to declare the propagation engines, and generated two FlatZinc files, one with $m = 10$ and the other with $m = 11$. Using the set of global constraints of Choco, the problem of size $m = 10$ is modeled with 55 variables and 57 constraints (one `allDifferent` constraint, 11 binary inequalities and 45 ternary equations), the search strategy is based on variables a, preserving the lexicographic ordering and choosing the lower bound of each decision variable.

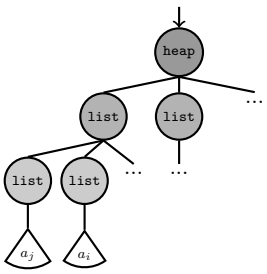
First, we introduce the declaration of propagation engines used to resolve the `Golomb ruler` problem. In addition to `prop`, `var` and `7qd` presented before, we declare `heap-var`, `heap-var-prio` and `2-coll`.

First of all, `heap-var` (Figure 10) is a variable oriented propagation engine based on a heap, as described in [9]. Arcs are organized by variables into lists, then all lists are put into a heap. The list that contains the smallest cardinality variable is selected for propagation. The selection is repeated until the heap becomes empty. Then, we declare `heap-var-prio` (Figure 11), a variant of `heap-var` where arcs of a variable are sorted with respect to their priority. The idea is to execute first, for a variable, the lighter propagators. This should help in discriminating `AllDifferent` propagators and other binary and ternary propagators. Finally, we declare `2-coll` (Figure 12), a composition of two collections in a list. The first collection is based on `M`, arcs involving the mark variables a. They are put in a list sorted with respect

Fig. 11: Variant of the heap-based propagation engine.

```
1: All: true;
2: All as heap(wone) of {
3:     each var as list(for) of {
4:          each prop.priority as list(for)
6:          key any.prop.priority
5: } key any.any.var.card};
```

(a) Declaration.

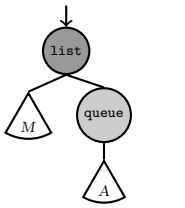


(b) Tree representation

Fig. 12: Two-collection propagation engine.

```
1: M: in(mark);
2: A: true;
3: list(wfor) of {
4:   list(wfor) of {M key var.name},
5:   queue(wone) of {A}
6: };
```

(a) Declaration.



(b) Tree representation

Table 2: Various engines for `Golomb ruler`.

Engine	$m = 10$		$m = 11$	
	`time` (sec.)	`pex` (10^6)	`time` (sec.)	`pex` (10^6)
`prop`	3.83	10.085	76.70	191.894
`var`	2.99	10.305	55.50	192.034
`7qd`	3.19	10.860	60.67	200.108
`heap-var`	2.79	10.717	53.09	196.525
`heap-var-prio`	3.31	11.242	59.84	207.973
`2-coll`	**2.69**	**9.975**	**52.48**	**185.921**

to their name in increasing order. Thus arcs involving a_i are treated before those on a_{i+1}. The `wfor` keyword ensures an iterative sweep propagation and a *local* fix-point (there is no pending events on arcs of `M`). The second collection is a queue that handles the remaining arcs (those involved in the differences variables) and ensures a local fix-point (`wone`).

Table 2 shows the results for the various declared propagation engines. It reports the solving time (`time` in seconds) and the number of propagator executions (`pex`) for $m = 10$ and $m = 11$. Even though, in some cases, there is no correlation between the number of propagator executions and the solving time (for instance `7qd` requires about 5.9% less checks than `prop` but takes 23.2% more time), efficient propagation engines run fewer executions. First, we observe the same result as [9]: the selection of the smallest cardinality variable allows us to efficiently solve the problem. Furthermore, discriminating propagators of a variable using priority, in `heap-var-prio`, does not seem necessary; this postpones `AllDifferent` propagator executions too late. Finally, the `2-coll` engine is the most efficient strategy: it

is very close to `heap-var` but it does not require evaluation of a dynamic criterion and should scale better. It is based on one `list` and one `queue` that have constant complexity operations.

But, more importantly, it is how prototyping has been simplified thanks to the DSL. Thrice we have been able to define, in a few lines, a specific behavior, with the help of variables, constraints and their properties, to produce a safe interpreted propagation engine, and to evaluate it in a given constraint solver. This simplifies the evaluation process and promotes the study of revision ordering within constraint satisfaction problems.

5 Conclusion and Future Work

Our first motivation, in this paper, was to provide a tool that simplifies propagation engine configuration within a constraint solver. First, we recalled the propagation algorithm variants (arc-, variable- and propagator-oriented), the widespread implementation choices in modern solvers and how revision ordering can be statically or dynamically adapted, using convenient propagation sets or groups of propagators. Then, we have presented a portable and extensible Domain Specific Language that enables concise, yet powerful, descriptions of propagation engine behaviors without any specific knowledge of a programming language. We have also listed the prerequisites to fully benefit from this DSL and the properties it provides. Then, we ensured that DSL-driven propagation engines satisfy the same guarantees as any native ones. Finally, we evaluated the overhead of parsing and executing DSL-driven propagation engines for prototyping. We showed the expressiveness of the DSL and the flexibility benefit of implementing it in a solver. In addition of commonly used propagation engine declarations, it enables the expression of dedicated propagation schemas.

Enabling propagation configuration also comes with limitations, and this paves the way for future work. First of all, engine definition in the DSL could be analyzed (detection of ill-formed engines, optimization with rewriting rules, etc.). We should also study how to adapt the declarations in presence of constraint reformulations: the main restriction to reformulation in the DSL is due to predicates and attributes related to group definition, and alternatives should be proposed. This would help to fully integrate our DSL into Constraint Programming standardization. Another future work is to study real-life problems with the help of the DSL, and its interaction with global constraints and decision strategies.

Acknowledgement

We would like to thanks the reviewers for their helpful remarks and comments on previous versions of this paper.

References

1. A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.
2. J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inf. Sci.*, 19(3):229–250, 1979.
3. R. Mohr and T. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, March 1986.
4. P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *AI*, 57(2-3):291–321, 1992.
5. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP '97*, pages 191–206, 1997.
6. F. Laburthe. Choco: Implementing a CP kernel. In *TRICS*, pages 71–85, 2000.
7. A. van Deursen, P. Klint and, J. Visser Domain-specific languages: an annotated bibliography In *SIGPLAN Not.*, pages 26–36, 2000.
8. A. López-Ortiz, CG. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI*, pages 245–250, 2003.
9. F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. In *CP'04*, pages 9–43, 2004.
10. C. Bessiere. Constraint propagation. Technical report 06020, LIRMM, 2006.
11. I. P. Gent, C. Jefferson, and I. Miguel. Minion: A fast scalable constraint solver. In *Proceedings of ECAI 2006*, pages 98–102. IOS Press, 2006.
12. I. P. Gent and C. Jefferson, and I. Miguel. Watched literals for constraint propagation in minion. In *Procedings CP 2006*, pages 182–197, 2006.
13. I. Katriel Expected-Case Analysis for Delayed Filtering In *Lecture Notes in Computer Science*, pages 119–125, Springer Berlin Heidelberg, 2006.
14. M. Z. Lagerkvist and C. Schulte. Advisors for incremental propagation. In *CP 2007*, pages 409–422, 2007.
15. I. P. Gent, I. Miguel, and P. Nightingale. Generalized arc consistency for the alldifferent constraint: An empirical survey. *Artificial Intelligence*, 172(18):1973–2000, 2008.
16. C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 2008.
17. M. Z. Lagerkvist and C. Schulte. Propagator groups. In Ian Gent, editor, *CP'09*, pages 524–538, 2009.
18. C. Schulte and G. Tack. Implementing efficient propagation control. In Christopher Jefferson, Peter Nightingale, and Guido Tack, editors, *TRICS 2010*, 2010.
19. I. P. Gent and T. Walsh. Csplib: a benchmark library for constraints. Technical report, APES-09-1999, 1999.
20. N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck and G. Tack MiniZinc: Towards a standard CP modelling language. In *CP'07*, pages 529–543, 2007.
21. IBM ILOG CPLEX Optimizer. http://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/, 2009.
22. Gecode: Generic Constraint Development Environment. http://www.gecode.org, 2012.
23. JaCoP: Java Constraint Solver. http://www.jacop.eu/, 2012.
24. or-Tools: Operations Research Tools developed at Google. http://code.google.com/p/or-tools/, 2012.