PRES LUNAM
Ecole Doctorale STIM
Sciences et Technologies de l'Information et Mathématiques

Spécialité : Informatique
Laboratoire : INRIA
Equipe : TASC

Propagation Engine Prototyping with a DSL

Prud'homme, Charles
Mél : charles.prudhomme@mines-nantes.fr

Résumé : Constraint propagation is at the heart of constraint solvers. Two main trends co-exist : variable-oriented propagation engines and constraint-oriented propagation engines. Those two approaches ensure the same consistency level, generally arc-consistency, but their efficiency (computation time) can be quite different depending on the instance solved. 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 engine prototypes.

Mots clés : *Constraint propagation, configuration, language*
Collaborations : INRIA ASCOLA.

1 Introduction

Constraint propagation which lies at the heart of constraint programming solvers has been extensively studied during the last decades [1, 13]. Several algorithms exist ; they are mainly represented by modern variants of AC enforcing algorithms [1, 2, 3]. For a given algorithm, several orientations exist. The widely used ones are based on variable orientation or constraint orientation. For instance, IBM CPO, Choco, Minion and or-tools use a variable-oriented algorithm, while Gecode, SICStus Prolog and JaCoP use a constraint-oriented algorithm.

Designing a propagation engine, and more generally a constraint solver, is made of choices and compromises to conjugate 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, the propagation engine tends 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 its resolution process without an in-depth knowledge about the underlying solver. The latter may want to fast prototype new propagation strategies before a complex phase of internal development.

We propose to go a step further and present a solver-independent language which enables to configure constraint propagation at the modeling stage. Thus, 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.

In the following, Section 2 recalls the fundamentals of constraint propagation, as well as the state-of-the-art improvements in constraint propagation algorithms. Section 3 is dedicated to the description of the DSL.

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 in a set of n variables $\mathcal{V}$, their associated domains $\mathcal{D}$, and a collection of m constraints $\mathcal{C}$. The domain $d_v \in \mathcal{D}$ associated to a variable $v \in \mathcal{V}$ defines a finite set of integer values v can be assigned to. A constraint $c \in \mathcal{C}$ is defined over a set of variables $V_c \subseteq \mathcal{V}$. A constraint c is equipped with one or more filtering algorithms, named *propagators*. A propagator $p \in \mathcal{P}$ is a function which removes values not consistent anymore from the

domains of its variables. Let $c \in \mathcal{C}$, P_c denotes the set of propagators of c and V_p the set of variables of p such that for all $v \in V_p, v \in V_c \wedge p \in P_c$; the set of propagators of a CSP is denoted $\mathcal{P}$.

Constraint programming is based on constraint propagation [8, 15]. Local deductions related to each propagator application force to reconsider the satisfiability of the CSP ; this is achieved thanks to the application of a propagation algorithm [1]. The propagation algorithm is a mechanism that schedules and executes *arcs* in order to reach the fix-point of the propagation phase and to recover the satisfiability of the CSP. An arc a is a pair $\langle p^a, v^a \rangle$ which associates a propagator p^a and a variable v^a, such that $v^a \in V_{p^a}$; $\mathcal{A}$ represents the set of arcs of a CSP. The propagation algorithm is triggered on a domain reduction (for instance, a decision is applied), and has two main steps. First, an arc a is removed from a set Q. This arc points out a propagator p^a to execute and a domain modified variable v^a. Second, the propagator p^a is executed, with d_{v^a} as input. This can produce two results. Either the propagator has detected a *failure*, for instance a domain becomes empty, and the algorithm stops. Or it has possibly scheduled new arcs in the set Q. These steps are run until the set Q becomes empty which means that a fix-point has been reached again.

The way Q is *oriented* can have an incidence on the way propagators will be implemented. The orientation is stated by the type of items Q handles. Alternatives to arcs are (a) to schedule variables [2], (b) to schedule propagators [4]. In the case of *variable-oriented* propagation engines (a), when a variable is removed from the set Q, its propagators are executed in a loop. In the case of *propagator-oriented* propagation engines (b), when a variable is modified, its propagators are scheduled in a loop. The spread of global constraints in constraint solvers has brought out the importance of carefully organizing propagator execution with respect to their complexity. Thus, it has been shown that it is worth carefully scheduling propagators of global constraints [5, 9, 10, 11, 13] These approaches are very similar : they aim at reducing the number of scheduling calls and increasing filtering efficiency.

In addition to those works on scheduling conditions, the *revision ordering* of elements to propagate has 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 fairly treats item. However, alternatives have been proposed [7, 13, 14].

Finally, unlike search strategies that are commonly available in the forms of portfolios or combinators [16], propagation engines are more likely imposed to users in constraint solvers. For these reasons, we propose to go a step further by introducing a Domain Specific Language (DSL) to prototype a propagation algorithm, in a fast and robust manner.

3 A DSL to describe Propagation Engines

In order to simplify the expression, or declaration, of a propagation engine for a user, we introduce a *Domain Specific Language* [6].

3.1 Domain Specific Language

Considering $\mathcal{A}$ as an input, the objective of the DSL is to describe a global ordering over it. The DSL is divided in two parts (Figure 1) : first, arc to group assignment, second, propagation engine structure description based upon groups. This two-step process enables to first focus on arc properties, and then to compose and combine groups together in a propagation engine. For the sake of clarity, each part will be discussed individually.

FIGURE 1 – The DSL entry point.

$\langle propagation\_engine \rangle$: := $\langle group\_decl \rangle$+ $\langle structure\_decl \rangle$;

The syntax is presented in standard BNF adopting the following conventions : typewriter `tt` indicates a terminal ; italic $\langle it \rangle$ 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 $e\star$ 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.

3.1.1 Group assignment declaration

We now present the part of DSL dedicated to group declaration. Building groups of arcs is made with respect to assignment rules (depicted in Figure 2(a)). The user only knows variables or constraints at

```
⟨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⟩
```

(a) Group definitions

```
⟨structure⟩    ::=  ⟨struct⟩ | ⟨struct_reg⟩
⟨struct⟩       ::=  ⟨coll⟩ of {⟨elt⟩,...} [key ⟨comb_attr⟩]
⟨struct_reg⟩   ::=  ⟨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
```

(b) Data structure of the propagation engine

modeling. Properties listed here are present in many constraint solvers. Propagators are only referenced through theirs properties because they are not accessible as modeling objects.

A group is declared with the help of 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 indicates a group membership. A predicate is said to be *in extension* if its declaration is based on a variable or a constraint (pointed out with `var_id` and `cstr_id`). Arcs associated with the variable or the constraint will be candidate for the group. A predicate is said to be *in intension* if its declaration is based on one or more properties. Any arcs of the resulting set must satisfy the properties. Intension predicates are built on three parameters : 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). Predicates can be combined together with the three boolean operations `&&`, `||` and `!`.

3.1.2 Structure declaration

The second part of the DSL (depicted in Figure 2(b)) covers the data structure description based upon groups of arcs. The objective here is twofold : indicating "where" to store an arc and "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 structure (ADT) and is a collection of other ADT and arcs. An ADT defines how its items will be scheduled and removed for revision. There are three of them : a `queue`, a `list` and a `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 (default comparison function is $\leq$, the keyword `max` changes it to $\geq$).

Basic iterators `one` and `wone` are defined for any ⟨*coll*⟩. `one` traverses one element from a ⟨*coll*⟩. `wone`, short version of 'while one', calls `one` until all elements are traversed. A `list` comes with two extra iterators : `for` and `wfor`. `for` sequentially traverses the list in the increasing order of an index. `for` does not enable going backwards into the list nor interrupting the traversing. `wfor`, short version of 'while for', calls `for` until all elements are traversed.

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 warranty a ⟨*coll*⟩ handles exclusively 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 2(b). An ⟨*ext_attr*⟩ enables to reach any item's attribute in a collection of items, such as its `size`.

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. It must be a ⟨*struct*⟩ or a ⟨*struct_reg*⟩.

A ⟨*struct*⟩ defines a collection ⟨*coll*⟩ which is composed of elements ⟨*elt*⟩. An ⟨*elt*⟩ can either reference a group identifier and an ordering instruction (⟨*id*⟩ [`key` ⟨*attribute*⟩]) or another ⟨*structure*⟩. A ⟨*struct*⟩ is defined in *extension*.

A ⟨*struct_reg*⟩ defines a regular structure. The expressivity implied by its regularity justifies its presence : it enables a compact expression of nested structures, where ⟨*struct*⟩ would require more verbose expression and more likely introduce bugs. A ⟨*struct_reg*⟩ is defined in *intension*. 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*⟩. The assignment to the right collection may be achieved dynamically during the resolution process when using a dynamic criterion.

A Domain Specific Language comes with properties, but also should ensure guarantees. We ensure DSL-driven propagation engines satisfy the same guarantees as any native ones.

4 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 an open and extensible Domain Specific Language that enables simple, yet powerful, descriptions of propagation engine behaviors without specific knowledge of a programming language. A future work is to study real-life problems with the help of the DSL, and its interaction with global constraints and decision strategies.

Références

[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] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A generic arc-consistency algorithm and its specializations. *AI*, 57(2-3) :291–321, 1992.

[4] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP '97*, pages 191–206, 1997.

[5] F. Laburthe. Choco : Implementing a CP kernel. In *TRICS*, pages 71–85, 2000.

[6] A. van Deursen, P. Klint and, J. Visser Domain-specific languages : an annotated bibliography In *SIGPLAN Not.*, pages 26–36, 2000.

[7] F. Boussemart, F. Hemery, and C. Lecoutre. Revision ordering heuristics for the constraint satisfaction problem. In *CP'04*, pages 9–43, 2004.

[8] C. Bessiere. Constraint propagation. Technical report 06020, LIRMM, 2006.

[9] 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.

[10] I. P. Gent and C. Jefferson, and I. Miguel. Watched literals for constraint propagation in minion. In *Procedings CP 2006*, pages 182–197, 2006.

[11] M. Z. Lagerkvist and C. Schulte. Advisors for incremental propagation. In *CP 2007*, pages 409–422, 2007.

[12] 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.

[13] C. Schulte and P. J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 2008.

[14] M. Z. Lagerkvist and C. Schulte. Propagator groups. In Ian Gent, editor, *CP'09*, pages 524–538, 2009.

[15] C. Schulte and G. Tack. Implementing efficient propagation control. In Christopher Jefferson, Peter Nightingale, and Guido Tack, editors, *TRICS 2010*, 2010.

[16] IBM ILOG CPLEX Optimizer. http ://www-01.ibm.com/software/integration/optimization/cplex-cp-optimizer/, 2009.