Enhancing Constraint Propagation with Composition Operators [⋆]

Laurent Granvilliers and Eric Monfroy

Computer Science Research Institute (IRIN)
University of Nantes
44322 Nantes cedex 3, France

Abstract. Constraint propagation is a general algorithmic approach for pruning the search space of a CSP. In a uniform way, K. R. Apt [1] has defined a computation as an iteration of reduction functions over a domain. In [2] he has also demonstrated the need for integrating static properties of reduction functions (commutativity and semi-commutativity) to design specialized algorithms such as `AC3` and `DAC`. We introduce here a set of operators for modeling compositions of reduction functions. Two of the major goals are to tackle parallel computations, and dynamic behaviours (such as slow convergence).

1 Introduction

A Constraint Satisfaction Problem (CSP) is defined by a set of domains, a set of variables, and a set of constraints. Solving a CSP consists in finding assignments of variables (i.e., values from their domains) that satisfy constraints. Since this problem is NP-hard, preprocessing techniques have been implemented to prune domains (search space) before backtracking, e.g., filtering algorithms based upon consistency properties of subsets of constraints [10]. Constraint propagation is a generic term for these techniques.

Recently, K. R. Apt [1] has proposed a unified framework for constraint propagation. The solving process is defined as a chaotic iteration [4], i.e., an iteration of reduction functions over domains. Under well-chosen properties of domains (e.g., partial ordering, well-foundedness) and functions (e.g., monotonicity, contractance), iteration-based algorithms are shown to be finite and confluent. Further refinements have been devised in [2] to tackle strategies based on additional properties of functions (such as `AC3` with idempotence and commutativity). Hence, specializations of component functions of the generic iteration algorithm have been implemented to tune the order of applications of functions.

In this paper we introduce new properties of functions (i.e., strongness, independency, and redundancy) to tackle parallel computations [6], and dynamic behaviours such as slow convergence [9]. Furthermore, the dynamic nature of

[⋆] This work has been supported by the IST project COCONUT from the European Community. An extended version of this paper will be submitted to the CP'2001 conference. E-mails: {granvilliers,monfroy}@irin.univ-nantes.fr such strategies led us to define the notion of composition operator. Basically, a composition operator is a local strategy for applying functions: it implements combinators for sequential or parallel computation, and for computation of closures. An iteration (preserving the semantics —termination, confluence—) is then defined as a sequence of application of composition operators. This approach provides flexibility (definition of components for creating composition operators) and generality (single iteration algorithm).

Using composition operators, we then present several strategies for modeling heuristics and properties of well-known constraint solvers: priorities of constraints in the finite domain solver `Choco` [8], slow convergence arising in interval narrowing [9], efficient heuristics for interval narrowing [5] as is done in `Numerica` [12], and finally, parallel computations [6]. Essentially, each strategy dynamically creates and applies composition operators until a fixed-point is computed.

The outline of this paper is the following. A basic constraint propagation framework is introduced in Section 2. Our new framework based on composition operators is described in Section 3. In Section 4, some existing strategies are shown to fit in our model.

2 Constraint Propagation

Our aim is not to be as general as in [2]. We consider here contracting and monotonic functions on a finite semilattice ordered by set inclusion. Such a domain is used to address the decoupling of reductions arising in parallel computations (see Section 3). Note that the results given here hold under these assumptions.

The *computation domain* is a finite semilattice $(\mathcal{D}, \subseteq, \cap)$, i.e., a partially ordered set $(\mathcal{D}, \subseteq)$ in which every nonempty finite subset has a greatest lower bound (an inf element). The ordering $\subseteq$ corresponds to set inclusion. The meet operation is the set intersection $\cap$, and the bottom element is the empty set. Note that in most constraint solvers, the computation domain is a finite lattice, i.e., finite domains such as Booleans, Integers, floating-point intervals.

Definition 1 (Reduction function). *Consider a function f on $\mathcal{D}$.*

- *f is contracting if $\forall x \in \mathcal{D},\ f(x) \subseteq x$;*
- *f is monotonic if $\forall x, y \in \mathcal{D},\ x \subseteq y \Rightarrow f(x) \subseteq f(y)$.*

A reduction function on $\mathcal{D}$ *is a contracting and monotonic function on $\mathcal{D}$.*

In the following we consider a finite set of reduction functions $F = \{f_1, \ldots, f_k\}$ on $\mathcal{D}$.

Definition 2 (Iteration). *Given an element $d \in \mathcal{D}$, an iteration of F over d is an infinite sequence of values $d_0, d_1, \ldots$ defined inductively by:*

$$\begin{aligned} d_0 &:= d \\ d_i &:= f_{j_i}(d_{i-1}) \quad i \geqslant 1 \end{aligned}$$

where j_i is an element of $[1, \ldots, k]$. A sequence $d_0 \supseteq d_1 \supseteq \cdots$ of elements from $\mathcal{D}$ stabilizes *at e if for some $j \geqslant 0$ we have $d_i = e$ for $i \geqslant j$.*

We have the following stabilization lemma.

Lemma 1. *Suppose that an iteration of F over d stabilizes at a common fixed-point e of the functions from F. Then, e is the greatest common fixed-point of the functions from F that is included in d, i.e., $(\cap_{i=1}^{k} f_i) \uparrow \omega(d)$.*

Proof. See [2]. It follows from the monotonicity of the reduction functions.

The generic iteration algorithm (`GI`) defined by K. R. Apt in [2] is given in Table 1. The correctness of `GI` is stated by Theorem 1.

Table 1. Generic Iteration Algorithm based on Reduction Functions.

```
GI ($F$: set of reduction functions on $\mathcal{D}$ ; $d$: element of $\mathcal{D}$): $\mathcal{D}$
begin
    $G$ := $F$
    while $G \neq \varnothing$ do
        choose $g \in G$
        $G$ := $G - \{g\}$
        $G$ := $G \cup$ update $(G, g, d)$
        $d$ := $g(d)$
    od
    return $d$
end
```

where for all G, g, d the set of functions `update` (G, g, d) is such that
A. $\{f \in F - G \mid f(d) = d \ \wedge \ fg(d) \neq g(d)\} \subseteq$ `update` (G, g, d)
B. $g(d) = d$ implies that `update` $(G, g, d) = \varnothing$
C. $gg(d) \neq g(d)$ implies that $g \in$ `update` (G, g, d)

Theorem 1. *Every execution of `GI` terminates and computes in d the greatest common fixed-point of the functions from F.*

Proof. See [2]. It follows from the monotonicity and contractance of the reduction functions, and the well-foundedness of the ordering of the semilattice.

3 Composition Operators

We restrict our attention to simple domains. The generalization of our results to compound domains, i.e., considering k-ary functions on $(\mathcal{D}_1, \ldots, \mathcal{D}_k)$, is straightforward.

3.1 Definitions

Consider a finite set of functions $F = \{f_1, \ldots, f_k\}$ on $\mathcal{D}$. We introduce a set of composition operations on F as follows:

$$
\begin{array}{rlll}
\text{Sequence:} & F^{\circ} & \text{denotes the function} & x \mapsto f_1 f_2 \ldots f_k(x) \\
\text{Closure:} & F^{\omega} & & x \mapsto (\cap_{i=1}^{k} f_i) \uparrow \omega(x) \\
\text{Decoupling:} & F^{\cap} & & x \mapsto f_1(x) \cap \cdots \cap f_k(x)
\end{array}
$$

Note that F is supposed to be ordered since the sequence operation is not commutative. This assumption is no longer necessary for the closure and decoupling operations since the computation of a fixed-point is a declarative property, and the intersection operation is commutative.

There are several motivations for introducing such operations in a generic propagation framework:

– The sequence operation fixes the order of application of the reduction functions. It can be used for computing directional arc consistency based on the semi-commutativity property, for modeling priorities of solvers, and for implementing heuristics or knowledge of solvers about their relative efficiencies.
– The closure operation allows us to make a closure from a non idempotent function, and to describe multi-level algorithms that compute fixed-points of different solvers.
– The decoupling operation is essentially used to model parallel computations, enforcing different functions over the same domain, and then computing the intersection using a fold reduction step.

The notion of composition operator models a function (a complex solver) built from composition operations.

Definition 3 (Composition operator). *Let $F = \{f_1, \ldots, f_k\}$ be a finite set of reduction functions. A* composition operator *on F is a function $\mathcal{D} \to \mathcal{D}$ defined by induction as follows:*

Atomic: f_i *is a composition operator for* $i = 1, \ldots, k$.

Given a finite set Φ of composition operators on F,

Sequence: Φ° *is a composition operator;*
Closure: Φ^{ω} *is a composition operator;*
Decoupling: $\Phi^{\cap}$ *is a composition operator.*

The generator $\texttt{Gen}(\phi)$ *of a composition operator ϕ on F is the subset of functions from F that are "involved" in ϕ. It is defined inductively by:*

– $\texttt{Gen}(\phi) = \{\phi\}$ *if ϕ is an atomic operator from F;*
– $\texttt{Gen}(\phi) = \texttt{Gen}(\phi_1) \cup \cdots \cup \texttt{Gen}(\phi_k)$ *if* $\phi = \{\phi_1, \ldots, \phi_k\}^{\star}$ *for* $\star \in \{\circ, \omega, \cap\}$.

Lemma 2 states that a composition operator is also a reduction function.

Lemma 2. *Consider a finite set of reduction functions $F = \{f_1, \ldots, f_k\}$. Then, every composition operator on F is*

(i) contracting;
(ii) monotonic.

Proof. The proof is done by induction. Every atomic operator is contracting and monotonic by definition of a reduction function. Now consider a set $\Phi = \{\phi_1, \ldots, \phi_k\}$ of contracting and monotonic composition operators.

(i) By hypothesis the composition operators from Φ are contracting. Then it is immediate to prove the contractance of Φ°, Φ^ω, and $\Phi^\cap$.
(ii) Given $x, y \in \mathcal{D}$ suppose that $x \subseteq y$.
  – Since every ϕ_i is supposed to be monotonic, then, we have $\phi_k(x) \subseteq \phi_k(y)$, and then, $\phi_{k-1}\phi_k(x) \subseteq \phi_{k-1}\phi_k(y)$, and so on. It follows that Φ° is monotonic.
  – To prove the monotonicity of Φ^ω we consider the function $\varphi : \cap_{i=1}^k \phi_i$. Then, we prove that φ is monotonic (third item). It follows that $\varphi \uparrow \omega(x) \subseteq \varphi \uparrow \omega(y)$. Then it is immediate to prove (by a double inclusion) that the set of fixed-points of φ coincides with the set of common fixed-points of the functions from Φ. As a consequence, we have $\varphi \uparrow \omega \equiv \Phi^\omega$, that completes the proof.
  – For $i = 1, \ldots, k$, we have $\phi_i(x) \subseteq \phi_i(y)$ by monotonicity of ϕ_i. It follows that $\cap_{i=1}^k \phi_i(x) \subseteq \cap_{i=1}^k \phi_i(y)$, that ends the proof.

Lemma 3 states that a fixed-point of a composition operator is a common fixed-point of the functions from its generator. The key idea is that the application of a composition operator implies that each reduction function in its generator is applied at least once.

Lemma 3. *Consider a finite set of reduction functions F and a composition operator ϕ on F. Then, e is a fixed-point of ϕ if and only if e is a common fixed-point of the functions from $\texttt{Gen}(\phi)$.*

Proof. We prove by induction the equivalence $\phi(e) = e \Leftrightarrow \forall f \in \texttt{Gen}(\phi)\ f(e) = e$. It obviously holds for an atomic operator ϕ. Now consider a set of composition operators $\Phi = \{\phi_1, \ldots, \phi_k\}$ on F and assume that the equivalence holds for each $\phi_1, \ldots, \phi_k$. If we have $\phi \equiv \Phi^\star$ for $\star \in \{\circ, \omega, \cap\}$, then it follows:

$$\begin{aligned} \phi(e) = e &\Leftrightarrow \forall i \in \{1, \ldots, k\}\ \phi_i(e) = e && \textit{(immediate result)} \\ &\Leftrightarrow \forall i \in \{1, \ldots, k\}\ \forall f \in \texttt{Gen}(\phi_i)\ f(e) = e && \textit{(induction hypothesis)} \end{aligned}$$

Considering $\texttt{Gen}(\phi) = \cup_i \texttt{Gen}(\phi_i)$ completes the proof.

The notion of iteration is slightly extended to deal with composition operators instead of reduction functions.

Definition 4 (Iteration). *Consider a finite set of reduction functions F, and a finite set $\Phi = \{\phi_1, \ldots, \phi_k\}$ of composition operators on F. Given an element $d \in \mathcal{D}$, an iteration of Φ over d is an infinite sequence of values $d_0, d_1, \ldots$ defined inductively by:*

$$\begin{aligned} d_0 &:= d \\ d_i &:= \phi_{j_i}(d_{i-1}) \quad i \geqslant 1 \end{aligned}$$

where j_i is an element of $[1, \ldots, k]$.

Lemma 1 (stabilization lemma from K. R. Apt) remains valid, that follows from the monotonicity of the composition operators (Lemma 2). Moreover, we have the following, essential result.

Lemma 4. *If an iteration on $\Phi = \{\phi_1, \ldots, \phi_k\}$ over d stabilizes at a common fixed-point e of the functions from Φ, and $F = \cup_i \mathtt{Gen}(\phi_i)$, then $e = F^\omega(d)$.*

Proof. *We first prove that e is a common fixed-point of the functions from F. By hypothesis e is a fixed-point of each $\phi_i \in \Phi$. By Lemma 3 it follows that $f(e) = e$ for each $f \in \mathtt{Gen}(\phi_i)$. The proof is completed since by hypothesis the set of generators covers F.*

We prove now that e is the greatest common fixed-point of the functions from F. Consider a common fixed-point e' of the functions from F. It suffices to prove that e' is included in every element from the iteration, namely $d_0, d_1, \ldots$. It obviously holds for $i = 0$. Suppose now it holds for i, i.e., $e' \subseteq d_i$, and assume that $d_{i+1} = \phi_j(d_i)$ for some $j \in [1, \ldots, k]$. By monotonicity of ϕ_j we have $\phi_j(e') \subseteq \phi_j(d_i)$. By Lemma 3 we have $\phi_j(e') = e'$, that completes the proof.

Table 2. Generic Iteration Algorithm based on Composition Operators.

```
GIco($F$: set of reduction functions on $\mathcal{D}$ ; $d$: element of $\mathcal{D}$): $\mathcal{D}$
begin
    $G$ := $F$
    while $G \neq \varnothing$ do
        $\phi$ := create a composition operator on $G$
        $G$ := $G - \mathtt{Gen}(\phi)$
        $G$ := $G \cup \mathtt{update}\,(G, \phi, d)$
        $d$ := $\phi(d)$
    od
    return $d$
end
```

where for all G, ϕ, d the set of functions $\mathtt{update}\,(G, \phi, d)$ is such that

A. $\mathtt{upA} := \{f \in F - G \mid f(d) = d \ \wedge \ f\phi(d) \neq \phi(d)\} \subseteq \mathtt{update}\,(G, \phi, d)$

B. $\mathtt{upB} := \phi(d) = d$ implies that $\mathtt{update}\,(G, \phi, d) = \varnothing$

C. $\mathtt{upC} := \{f \in \mathtt{Gen}(\phi) \mid f\phi(d) \neq \phi(d)\} \subseteq \mathtt{update}\,(G, \phi, d)$

We describe now a generic iteration algorithm `GIco` based on composition operators on a finite set of reduction functions F (see Table 2). Note that the set of composition operators is not fixed, since each operator is dynamically created from the set G of active reduction functions, and it is applied only once. Nevertheless, Theorem 2 proves the correctness of `GIco` with respect to F.

Theorem 2. *Every execution of* `GIco` *terminates and computes in* d *the greatest common fixed-point of the functions from* F*.*

Proof. The proof is a direct adaptation of Apt's [2]. To prove termination it suffices to prove that the pair $(d, \#G)$ *strictly decreases in some sense at each iteration of the* `while` *loop, and to note that the ordering* $\subseteq$ *is well-founded.*

The correctness is implied by the invariant of the `while` *loop, i.e., every* $f \in F - G$ *is such that* $f(d) = d$*. It follows that the final domain is a common fixed-point of the functions from* F *(since* $G = \varnothing$*). The second part of the proof of Lemma 4 ensures that it is the greatest one included in the initial domain.*

The following corollary concerns the application of a closure operator in algorithm `GIco`.

Corollary 1. *Consider operator* ϕ *that is applied in algorithm* `GIco`*. If* ϕ *is idempotent, then assumption C is reduced to* `upC := ∅`*.*

Proof. See Apt [2].

3.2 On Properties of Functions

In this section we examine the properties of reduction functions that are preserved by compositions. We first define a set of properties of interest for our purpose.

Definition 5. *Consider two reduction functions* f, g *on* $\mathcal{D}$*. Then, for all* $x \in \mathcal{D}$*:*

- f *is idempotent if* $ff(x) = f(x)$
- f *and* g *commute if* $fg(x) = gf(x)$
- f *semi-commutes with* g *if* $gf(x) \subseteq fg(x)$
- f *is stronger than* g *if* $f(x) \subseteq g(x)$
- f *and* g *are independent if* $fg(x) = gf(x) = f(x) \cap g(x)$
- f *and* g *are redundant if* $f(x) = g(x)$
- f *and* g *are weakly redundant if* $f \uparrow \omega(x) = g \uparrow \omega(x)$

The property of idempotence of a composition operator allows us to modify assumption C on the `update` function of the `GIco` algorithm (see Corollary 1). For this purpose some kinds of composition operators are shown to be idempotent in the following proposition.

Proposition 1. *Consider a set of composition operators* $\Phi = \{\phi_1, \ldots, \phi_k\}$*. Then,* ϕ *is idempotent if:*

i) ϕ *is a closure operator* Φ^ω*.*

ii) *ϕ is a sequence operator Φ°, ϕ_i semi-commutes with ϕ_j for $i > j$, and each ϕ_i is idempotent.*

iii) *ϕ is a sequence operator Φ°, for each i there exists $j < i$ such that ϕ_j is stronger than ϕ_i, and ϕ_1 is idempotent.*

iv) *ϕ is a decoupling operator $\Phi^\cap$, each ϕ_i is idempotent, and the ϕ_i are pairwise independent.*

Proof. *It suffices to prove that in all these cases, ϕ is idempotent. The proof is then completed by Lemma 3, i.e., each element computed by ϕ is a common fixed-point of the functions from its generator.*

i) *The proof is obvious.*

ii) *See Apt [2].*

iii) *Since the relation of strongness is transitive, then ϕ_1 is stronger than ϕ_i for $i \in [2, \ldots, k]$. Now it suffices to prove that $\phi_j \phi_1 \ldots \phi_k(x) = \phi_1 \ldots \phi_k(x)$. We have*

$$\phi_1(y) = \phi_1\phi_1(y) \subseteq \phi_j\phi_1(y) \subseteq \phi_1(y)$$

since ϕ_1 is idempotent, ϕ_1 is stronger than ϕ_j, and ϕ_j is contracting. This ends the proof if we set $y = \phi_2 \ldots \phi_k(x)$.

iv) *Let us prove that function $\cap_{i=1}^{k}\phi_i$ is idempotent. We prove by induction on j that $\cap_{i=1}^{j}\phi_i$ is idempotent for $j = 1, \ldots, k$ and that it is independent on ϕ_{j+1} for $j = 1, \ldots, k-1$. It holds for $j = 1$ since by hypothesis, ϕ_1 is idempotent, and ϕ_1 and ϕ_2 are independent. Now fix $1 < j < k$, and consider that $\varphi : \cap_{i=1}^{j}\phi_i$ is idempotent, and that φ and ϕ_{j+1} are independent. We prove that function $\psi : \varphi \cap \phi_{j+1}$ is idempotent and independent on ϕ_{j+2}. Given an element $x \in \mathcal{D}$, we have:*

$$\begin{aligned}
\psi(\psi(x)) &= \varphi(\varphi(x) \cap \phi_{j+1}(x)) \cap \phi_{j+1}(\varphi(x) \cap \phi_{j+1}(x)) && \\
&= \varphi\varphi(\phi_{j+1}(x)) \cap \phi_{j+1}\phi_{j+1}(\varphi(x)) && \textit{independence } \varphi, \phi_{j+1} \\
&= \varphi\phi_{j+1}(x) \cap \phi_{j+1}\varphi(x) && \textit{idempotence } \varphi, \phi_{j+1} \\
&= (\varphi(x) \cap \phi_{j+1}(x)) \cap (\phi_{j+1}(x) \cap \varphi(x)) && \textit{independence } \varphi, \phi_{j+1} \\
&= \psi(x) && \textit{commutativity of } \cap
\end{aligned}$$

Then ψ is idempotent. Now we prove that ψ and ϕ_{j+2} are independent, i.e.,

$$(\varphi \cap \phi_{j+1})(\phi_{j+2}(x)) = \phi_{j+2}((\varphi \cap \phi_{j+1})(x)) = (\varphi \cap \phi_{j+1})(x) \cap \phi_{j+2}(x)$$

It suffices to prove that each term of this formula is equivalent to $\cap\phi_{i=1}^{j+2}(x)$. It obviously holds for the last term. For the first term, we have:

$$\begin{aligned}
(\varphi \cap \phi_{j+1})(\phi_{j+2}(x)) &= \varphi\phi_{j+2}(x) \cap \phi_{j+1}\phi_{j+2}(x) \\
&= \phi_1\phi_{j+2}(x) \cap \cdots \cap \phi_j\phi_{j+2}(x) \cap \phi_{j+1}\phi_{j+2}(x)
\end{aligned}$$

The independence of each pair (ϕ_i, ϕ_j) ends the proof. For the second term we have:

$$\phi_{j+2}((\varphi \cap \phi_{j+1})(x)) = \phi_{j+2}\varphi(x) \cap \phi_{j+2}\phi_{j+1}(x)$$

It suffices to remark that ϕ_{j+2} and ϕ_{j+1} are independent by hypothesis, and then to prove that ϕ_{j+2} and φ commute, i.e., $\phi_{j+2}\varphi(x) = \varphi\phi_{j+2}(x)$. This result is easily proved by induction since

$$\phi_{j+2}\varphi(x) = \phi_{j+2}\phi_1 \dots \phi_j(x)$$

by independence of each pair (ϕ_i, ϕ_j), and then

$$\phi_{j+2}\phi_1 \dots \phi_j(x) = \phi_1\phi_{j+2}\phi_2 \dots \phi_j(x)$$

by independence of ϕ_{j+2} and ϕ_1, and so on.

In the following proposition, we identify cases where the computation of a closure of a set of composition operators can be improved, according to properties of independence and redundancy of operators.

Proposition 2. *Consider a set of composition operator Φ and a composition operator φ. Then, the following properties hold:*

1. *If for all $\phi \in \Phi$, φ and ϕ are independent, then $(\Phi \cup \{\varphi\})^\omega = \Phi^\omega \cap \varphi \uparrow \omega$.*
2. *If there exists $\phi \in \Phi$ such that φ and ϕ are weakly redundant, then $(\Phi \cup \{\varphi\})^\omega = \Phi^\omega$.*

Proof. 1. *It suffices to show that operator $\theta : \Phi^\omega \cap \varphi \uparrow \omega$ is idempotent, i.e., $\theta\theta(x) = \theta(x)$ for all $x \in \mathcal{D}$. The proof then follows by Lemma 4. Given a particular element x, consider that $\theta(x) = \phi_{i_1} \dots \phi_{i_k}(x) \cap \varphi \dots \varphi(x)$, and that $\theta\theta(x) = \phi_{j_1} \dots \phi_{j_l}\theta(x) \cap \varphi \dots \varphi\theta(x)$. A simple induction, using the hypothesis of independence, then allows us to rewrite $\theta\theta(x)$ as $\theta(x) \cap \theta(x)$.*
2. *The proof is obvious since by definition, Φ^ω is a fixed-point of ϕ. Hence, it is also a fixed-point of φ.*

3.3 From Decomposition to Composition

A glass-box solver mainly combines elementary solving components. The relation between solvers and components is expressed by Definition 6.

Definition 6 (Decomposition of a function). *Consider a reduction function f on $\mathcal{D}$. A finite set of reduction functions F on $\mathcal{D}$ is a* decomposition *of f if $\#F > 1$ and for every $x \in \mathcal{D}$, we have $F^\omega(x) = f \uparrow \omega(x)$.*

For instance consider a filtering algorithm enforcing a local consistency technique over a CSP. The generic computation is an iteration such that the consistency of a set of constraints, a constraint, or a constraint projection is verified at each step. Note that there is a correspondence between the decomposition of a data (e.g., a CSP) in a set of elementary components (e.g., constraints), and the decomposition of a function (e.g., a solver associated with a CSP) in a conjunction of elementary functions (e.g., an elementary solver associated with constraints).

Considering different levels of granularity in a constraint solving process must not influence the semantics of computation. To this end we have the following result.

Proposition 3. *Consider a finite set of reduction functions $F = \{f_0, f_1, \ldots, f_k\}$ on $\mathcal{D}$. Let G be a decomposition of f_0. Given $H = G \cup \{f_1, \ldots, f_k\}$, we have $H^\omega(x) = F^\omega(x)$ for every $x \in \mathcal{D}$.*

Proof. The proof is very similar to the one of Lemma 1.

The decomposition of reduction functions leads to the notion of decomposition relation. It can be useful to compare levels of granularity of iteration algorithms.

Definition 7 (Decomposition relation). *Consider F, G two finite sets of reduction functions on $\mathcal{D}$. G is a* decomposition of F *if $\#G > \#F$ and $G^\omega(d) = F^\omega(d)$ for every $d \in \mathcal{D}$.*

Note that the decomposition relation is a strict partial order.

In practice the aim is to fastly design efficient algorithms. We believe that these requirements can be achieved in a glass-box and generic (e.g., object oriented) programming approach.

- Developing powerful propagation techniques can be tackled by the `GIco` algorithm taking as input a well-chosen decomposition of solvers. The generation of composition operators during the iteration allows us to efficiently combine elementary solvers using the best strategy with respect to the knowledge or properties of reduction functions. Several efficient strategies, dynamic in essence, are described in Section 4.
- Efficient prototyping is achieved by a generic programming approach. In fact in the case a generic (constraint propagation) algorithm is re-used, only a (generic) piece of code for each kind of elementary solver has to be implemented, e.g., for a reduction function enforcing a local consistency property. In practice a main challenge is then to automatically generate the set of reduction functions of the decomposition given a generic implementation of a function and a CSP. This issue is discussed in the conclusion but a precise description is out of the scope of this paper.

4 Specialized Algorithms

The constraint propagation framework described in this paper allows us to tackle existing, efficient algorithms based on strategies and heuristics.

4.1 Scheduling of Constraints with Priorities

`Choco` is a constraint programming system for finite domains that has been developed by Laburthe et al. [8]. The core algorithm is constraint propagation whose main feature is to process constraints according to the complexity in time of their associated solving algorithms. Thus, primitive constraints are first processed, and then, global constraints with linear, quadratic complexity, and so on until a fixed-point is reached. In other words the computation is a sequence of application of closures, each closure being connected to the previous one by means of propagation events (modification of domains).

Implementing the propagation engine of `Choco` using our generic algorithm `GIco` can be done by considering priorities of reduction functions. The creation of composition operators can be implemented in two ways as follows:

1. $\phi := \{g \in G \mid \mathit{priority}(g) = \alpha\}^{\omega}$ s.t. $\alpha = \min(\{\mathit{priority}(g) \mid g \in G\})$
2. $\phi := (G_1^{\omega} \cup \dots \cup G_p^{\omega})^{\circ}$ s.t. $G_1, \dots, G_p$ is a partition of G
   $\forall i \in [1, \dots, p], \forall g \in G_i, \text{priority}(g) = \alpha_i$
   $\alpha_p < \alpha_{p-1} < \dots < \alpha_1$

The first implementation describes the computation of the closure of the set of active reduction functions with the greatest priority (i.e., the computationally less expensive functions). It is a model of `Choco` in the sense that only functions with greatest priority are applied at a time. Note that the composition operator is a closure (and thus, idempotent), and consequently Corollary 1 applies for the `update` function (i.e., `upC`$= \varnothing$).

The second implementation is a sequence of closures, each closure processing the set of functions of a given priority. The main difference with respect to the first method is that no propagation step is performed between the application of two closures. This approach tends to minimize costs for updating propagation structures.

4.2 Sequentiality in Interval Constraints

The solver presented in [5] implements constraint propagation for interval domains, where reduction functions enforce box consistency for numeric constraints over the reals. It extends two existing solvers, namely `Numerica` [12] by Van Hentenryck et al., and the algorithm `BC4` [3] by Benhamou et al.

The solving process combines three kinds of reduction functions:

1. Function $f_{c,i}$ computes box consistency for the domain of variable x_i with respect to constraint c, i.e., $f_{c,i}$ is a projection function;
2. Function g_c implements constraint inversion for all variables of constraint c;
3. Function h computes a linear relaxation (by means of a first order Taylor approximation) for a constraint system, which is then processed by the interval Gauss-Seidel method.

The best strategy is based upon properties and heuristics: function $f_{c,i}$ is stronger than the projection of g_c on x_i, but they are redundant if x_i occurs only once in c; g_c is computationaly less expensive than functions $f_{c,i}$ for all i; h is in general more precise for tight domains, while the $f_{c,i}$ and g_c are more efficient for large domains. Using `GIco` this strategy is efficiently implemented as follows:

$$\phi := ((G - \{h\})^{\omega} \cup \{h\}^{\omega})^{\circ}$$

Note that h depends on all variables, that implies that h belongs to G.

Furthermore, the decomposition process that generates the set of reduction functions is tuned with respect to the redundancy property. Hence, each reduction function $f_{c,i}$ such that x_i occurs only once in c is removed (since g_c is as precise as $f_{c,i}$ for x_i). Proposition 2 guarantees that the output domain is consistent with respect to all constraints from the initial system.

4.3 Acceleration of Interval Narrowing

Interval narrowing, i.e., constraint propagation with interval domains, is inefficient if slow convergence happens. A slow convergence corresponds to a cycle of reduction functions $f_i \dots f_j \dots f_i$ such that each application only deletes a small part of a domain. When constraints are nonlinear constraints over the reals, this problem frequently arises, due to, e.g., singularities or points of tangence.

Lhomme et al. [9] have devised an efficient strategy based on cycle detection and simplification. The aim is to locally select and apply the best reduction functions while delaying some active functions supposed to slow the computation. Given a cycle, i.e., a set of functions G' from the propagation set G, the solving process using `GIco` can be described as follows:

1. G' is rewritten as $\Phi \cup \Phi'$ where Φ' contains all functions g from G such that for all $\phi \in G - \{g\}$, g and ϕ are independent;
2. Φ is rewritten as $\Phi_1 \cup \Phi_2$, Φ_1 being composed of the best functions from Φ. More precisely Φ_1 contains a function f per variable whose (current) domain can be modified by a function from Φ; the selected function f is the one that computes the largest reduction;
3. Doing so, the composition operator applied in `GIco` can be defined as:

$$\phi := (\Phi_1^{\omega} \cup \Phi_2)^{\circ}$$

   applying first the best functions, and then the ones that have been delayed because of the independency property. Note that each function from the set $(G - G') \cup \Phi_2$ has to be added in G by the `update` function.

4.4 Parallel Constraint Propagation

Parallel processing of numerical problems via interval constraints has been proposed as a general framework for high-performance numerical computation in [7]. Parallel constraint propagation [11] operationally consists in distributing reduction functions among processors, performing local computations, and then accumulating and intersecting new domains on some processors.

The decoupling composition operator can be used to implement parallel constraint propagation. A basic strategy is to create a partition of the propagation structure $G = G_1 \cup \cdots \cup G_k$, k being dependent on the number of processors, and to consider operator ϕ to be applied in the algorithm `GIco`:

$$\phi := G_1^{\circ} \cap \cdots \cap G_k^{\circ}$$

Moreover if one wants to perform more local computations before synchronisation and communication, a closure can be computed on each processor as follows:

$$\phi := G_1^\omega \cap \cdots \cap G_k^\omega$$

Nevertheless, it has been observed that the classical notion of parallel speed-up is not a correct measure of success for such algorithms. This is due to a parallel decoupling phenomenon: convergence may be faster when two interval contractions are applied in sequence than in parallel. As a consequence a parallel version of Lhomme's strategy, described in the previous section, has been proposed in [6]. Essentially, parallelism is only used to select the best functions, i.e., to create the decomposition $\Phi_1 \cup \Phi_2$ of the previous section.

Note that the decoupling phenomenon can be controlled according to the independence property of reduction functions. Proposition 2 ensures that a closure can be computed in parallel, provided that each subset of dependent functions is located on one processor. This property is not achievable in general for the whole CSP. Nevertheless the number of links between processors (corresponding to couples of dependent functions) can be minimized, which tends to maximize the amount of reductions of domains. In that case it may also be efficient to duplicate some functions on several processors in order to break some links. Further work will investigate such strategies, which have to be dynamic to guarantee load balancing.

5 Conclusion and Perspectives

A set of composition operations of reduction functions is introduced to design dynamic constraint propagation strategies. K. R. Apt's iteration model is slightly modified while preserving the semantics. Finally, several well-known strategies (using priorities of constraints, heuristics on the order of application of functions, and parallelism) are modeled using a single iteration algorithm.

A generic implementation of constraint propagation, integrating composition operators, has been designed. However it is out of scope of this article and it will be the topic of a second article.

The set of composition operators is (intentionally) reduced to sequence, closure, and decoupling operators. One may desire additional operators to model sequences of fixed length, quasi closures with a notion of precision, or conditional strategies with respect to dynamic criteria. We believe that their integration in our framework is feasible.

Acknowledgements

We are grateful to Frédéric Benhamou for interesting discussions on these topics.

References

1. Krzysztof R. Apt. The Essence of Constraint Propagation. *Theoretical Computer Science*, 221(1-2):179–210, 1999.

2. Krzysztof R. Apt. The Role of Commutativity in Constraint Propagation Algorithms. *ACM Transactions on Programming Languages and Systems*, 2001. To appear.
3. Frdric Benhamou, Frdric Goualard, Laurent Granvilliers, and Jean-Franois Puget. Revising Hull and Box Consistency. In *Proceedings of the 16th International Conference on Logic Programming*, pages 230–244, Las Cruces, USA, 1999. The MIT Press.
4. Daniel Chazan and Willard Miranker. Chaotic Relaxation. *Linear Algebra and its Applications*, 2:199–222, 1969.
5. Laurent Granvilliers. On the Combination of Interval Constraint Solvers. *Reliable Computing*, 2001. To appear.
6. Laurent Granvilliers and Gatan Hains. A Conservative Scheme for Parallel Interval Narrowing. *Information Processing Letters*, 74:141–146, 2000.
7. Gatan Hains and Maarten H. van Emden. Towards high-quality, high-speed numerical computation. In *Proceedings of Pacific Rim Conference on Communications, Computers and Signal Processing*, University of Victoria, B.C., Canada, 1997. IEEE.
8. Franois Laburthe and the OCRE project team. CHOCO: implementing a CP kernel. In *Proceedings of the CP'2000 workshop on techniques for implementing constraint programming systems*, Singapore, 2000.
9. Olivier Lhomme, Arnaud Gotlieb, and Michel Rueher. Dynamic Optimization of Interval Narrowing Algorithms. *Journal of Logic Programming*, 37(1–2):165–183, 1998.
10. Alan K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.
11. Eric Monfroy and Jean-Hugues Rty. Chaotic Iteration for Distributed Constraint Propagation. In *Proceedings of ACM Symposium on Applied Computing*, pages 19–24, San Antonio, Texas, USA, 1999. ACM Press.
12. Pascal Van Hentenryck, Laurent Michel, and Yves Deville. *Numerica: a Modeling Language for Global Optimization*. MIT Press, 1997.