Stratified Constructive Disjunction and Negation in Constraint Programming

Gotlieb, Arnaud
Simula Research Laboratory
P.O. Box 134
Lysaker, Norway
arnaud@simula.no

Marijan, Dusica
Simula Research Laboratory
P.O. Box 134
Lysaker, Norway
dusica@simula.no

Spieker, Helge
Simula Research Laboratory
P.O. Box 134
Lysaker, Norway
helge@simula.no

***Abstract*—Constraint Programming (CP) is a powerful declarative programming paradigm combining inference and search in order to find solutions to various type of constraint systems. Dealing with highly disjunctive constraint systems is notoriously difficult in CP. Apart from trying to solve each disjunct independently from each other, there is little hope and effort to succeed in constructing intermediate results combining the knowledge originating from several disjuncts. In this paper, we propose If Then Else (ITE), a lightweight approach for implementing stratified constructive disjunction and negation on top of an existing CP solver, namely SICStus Prolog `clp(FD)`. Although constructive disjunction is known for more than three decades, it does not have straightforward implementations in most CP solvers. ITE is a freely available library proposing stratified and constructive reasoning for various operators, including disjunction and negation, implication and conditional. Our preliminary experimental results show that ITE is competitive with existing approaches that handle disjunctive constraint systems.**

***Index Terms*—Constraint Programming, Constructive Disjunction, Stratified Reasoning, Implementation.**

I. Introduction

Constraint Programming (CP) is a powerful declarative programming paradigm where traditional instructions are replaced with relations over variables. CP has shown for a long term its ability to model and solve many hard combinatorial problems. This is due to the fact, that CP supports several types of relations between values and constraints, including arithmetical, numerical, logical, and symbolic constraints, as well as various computational domains, such as integers, reals, lists, strings, trees, graphs, etc. By considering a large variety of real-world problems, spanning from car manufacturing, configuration design to energy production and hardware and software engineering, CP is a crucial paradigm for exact solving of constraint systems, which has been proven in many practical applications.

However, unlike SAT-solving, CP has traditionally only considered conjunction of constraints. Even though powerful reasoning capabilities are available in CP, disjunctions have to be handled by the user, which has to find ways to model disjunctions using the available modeling tools and global constraints. In CP, the constraint system is a set of constraints implicitly combined via conjunction. Dealing with disjunctions in CP has always been perceived as challenging. This is due to the uneasy trade-off between search and inference while searching for satisfying assignments. Indeed, despite the usual exploitation of hypothesis refutation in the search process, inference has traditionally been mostly considered for conjunction of constraints. Notable exceptions include the definition of few global constraints, which embed disjunctive reasoning, such as `cumulative` [1] or `element` [10]. To illustrate the usual absence of inference for disjunctive constraints, consider the following request to SICStus Prolog `clp(FD)` [4], a state-of-the-art CP solver over Finite Domains:

Example 1:

```
Query: (X#=1) #\/ (X#=3) #\/ (X in 6..7)
Answer: X in inf..sup  // 'logical or' in clpfd
```

In this example, as no information is available on `X`, the solver uses only local reasoning for each disjunct and thus cannot perform any pruning. As a consequence, the domain of `X` is left unconstrained. Of course, for this request, an obvious and stronger answer would be the following:

Example 2:

```
Q: (X#=1) cd (X#=3) cd (X in 6..7)
A: X in{1}\/{3}\/(6..7)  //cd means 'constr. disj.'
```

Getting this result requires to use some global reasoning through a *constructive disjunction* operator, that is an operator able to construct knowledge from both disjuncts, without knowing which one will be true. In most CP implementations, the 'constructive disjunction' operator is unfortunately not natively available, as it is considered as too costly to propagate. Note that, besides the simple example given above, a complete implementation has to consider much more complex constraints in the disjuncts such as shown in the following example.

Example 3:

```
Q: A,B in 1..10, (A#>1,B#<9)cd(A#>2,A#<10),
   (A+7#=<B) cd ( cn(B+7#>A) )
```

This work is supported by the Research Council of Norway under the grant agreement Certus SFI.

```
// cn means 'constructive negation.'
A: A in{3}\/(8..10), B in(1..3)\/{10}
```

Constructive reasoning requires to perform a deep analysis of each disjunct before any constructive information can be propagated through the constraint network. Such constructive knowledge can be useful to solve efficiently many practical real-world problems originating from planning [6], scheduling [15], software engineering [5] or configuration. In many cases, real-world systems are highly configurable and each configuration can lead to a slightly different constraint system.

Constructive disjunction is not new in CP. Besides the initial proposition by Van Hentenryck [10], several implementations have been proposed in different CSP solvers including Oz [22], Gecode [19], SICStus Prolog [2] or Choco4 [17]. Implementation of constructive disjunction propagators in CP solvers is usually considered straightforward when it is perceived as an extension of constraint reification [11]. In that case, it suffices to associate a Boolean variable to any constraint, representing its truth value, and use Boolean constraints to combine the constraints. This approach is available in Choco4, for example, with its embedded SAT solver. In SICStus Prolog `clp(FD)`, besides the constraint reification mechanism, a more sophisticated approach is available through the `smt` global constraint [2] which propagates disjunctive linear information. Reasoning over disjunctions of temporal constraints has also been extensively considered in CP, with specialized algorithms [21].

The problem with the reification approach is that it propagates little constructive information. Entailment is usually poorly deductive as it is limited to local reasoning, that is, local filtering over the variables of the reified constraint. In most cases, more constructive knowledge can be propagated but the implementation of constructive disjunction requires a good tradeof between inference and search. An interesting attempt in that direction is given in [16], [20] where interval reasoning is used to propagate more information through constraint refutation. The proposed approach is however limited to numerical constraint systems and does not easily extend to finite domains constraints.

In this paper, we propose a lightweight approach to constructive disjunction which can be implemented on top of any existing CP solver, whatever be the level of filtering consistency considered and the constraint language in usage. We propose a parametric *stratified reasoning* to cope with the inherent combinatorial explosion of disjunctive problems. Our open-source and open-access implementation on top of the SICStus Prolog, called If Then Else (ITE)[1], makes use of the global constraint definition mechanism of `clp(FD)`. We argue that the availability of alternative ways to deal with disjunctive problems in CP is crucial to its adoption for solving real-world industrial problems, e.g., test case generation for specific domains [8] or generation of equiprobable test data [9].

The rest of this paper is organized as follows. Section II introduces the necessary background and notations to understand the rest of the paper. Section III presents constructive disjunction and negation and discusses of their implementation. This section also presents the various implementations of relaxation of entailments. It contains several examples to help the reader understand the proposed approach. Section IV proposes stratified reasoning to cope with possible combinatorial explosion. Section V gives our experimental results with ITE, while Section VI concludes this work and proposes further work.

II. Background

A. Constraint Satisfaction Problems

Constraint Satisfaction Problems (CSP) are materialized by $(\mathcal{X}, \mathcal{C})$ where $\mathcal{X} = \{X_1, \ldots, X_n\}$ stands for a set of n variables, each variable X_i taking a value into a finite domain D_i and, $\mathcal{C} = \{C_1, \ldots, C_m\}$ is a set of m constraints. A constraint C_k of arity r, is a relation over a subset of r variables from $\mathcal{X}$ which restrains the acceptable tuples for the relation. The subset of variables involved into the relation is usually known in advance, but there are cases where the size of the subset is parameterized by the problem instance. In the former case, the relations are called *primary constraints* while they are called *global constraints* in the latter. $C_k(X_j, \ldots, X_{j+r})$ is said to be *satisfied* by an assignment of its variables $X_j, \ldots, X_{j+r}$ to values $v_j, \ldots, v_{j+r}$ from their domain iff $C_k(X_j, \ldots, X_{j+r})$ evaluates to true with this assignment. A CSP is said to be *satisfiable* iff there exists at least one full assignment (i.e., an assignment of all variables in V also called a *solution*) such that all constraints in C are satisfied with this assignment. Otherwise, the CSP is said to be *unsatisfiable*. Note that, in this definition, the constraints of C are interpreted as a conjunction of constraints.

Solving a CSP means to either exhibit a solution or to prove unsatisfiability. In both cases, local filtering consistencies are used to approach this question. Among the many filtering consistencies proposed in the literature, we will focus only on two well-known, namely *hyperarc-consistency* and *bound-consistency*.

Definition 1: (hyperarc-consistency)
For a given CSP $(\mathcal{X}, \mathcal{C})$, a constraint C_k is *hyperarc-consistent* if for each of its variable X_i and value $v_i \in D_i$ there exist values $v_j, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{j+r}$ in $D_j, \ldots, D_{i-1}, D_{i+1}, \ldots, D_{j+r}$ such that $C_k(v_j, \ldots, v_{j+r})$ is satisfied. A CSP is hyperarc-consistent if for each of its constraints C_k, C_k is hyperarc–consistent.

Filtering a CSP with hyperarc-consistency can be very demanding in terms of computational resources. That is why other consistencies, less demanding, have been introduced. Among them, bound-consistency is a good compromise.

[1] Available at https://github.com/ite4cp/ite

Definition 2: (bound-consistency)
A constraint C_k is *bound-consistent* if for each variable X_i and value $v_i \in \{min(D_i), max(D_i)\}$ there exist values $v_j, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{j+r}$ in $D_j^*, \ldots, D_{i-1}^*, D_{i+1}^*, \ldots, D_{j+r}^*$ such that $C(v_j, \ldots, v_{j+r})$ is satisfied. In this definition, D_j^* stands for $min(D_j) \ldots max(D_j)$ (obviously $D_j^* \supseteq D_j$). A CSP $(\mathcal{X}, \mathcal{C})$ is *bound-consistent* if for every constraint C_k, C_k is bound-consistent.

More details on CSP and local filtering consistencies can be found in [12], [13], [18].

B. Global Constraint Definition

As said above, global constraints are relations over an unknown subset of variables. Typical examples include the `all_different`$(X_1, \ldots, X_n)$ constraints which constrains all variables to take different values (here, n is a user-defined parameter) or the `element`$(I, (X_1, \ldots, X_n), V)$ which constrains X_I to be equal to V. An important feature of modern CSP solvers is their ability to let the user define its own global constraints.

Defining a global constraint requires three elements to be given by the user:

1) **(Interface)** An interface has to be defined for the global constraint. This interface contains a name for the constraint along with a list of variables, possibly unbounded ;
2) **(Algorithm)** Each time the defined global constraint is considered in the propagation queue of the CSP solver, a filtering algorithm has to be launched with the goal to eliminate as much as possible inconsistent values of variables from their domain ;
3) **(Awakening)** Guidance to the awakening of the constraint in the propagation queue must also be provided. It is important to decide when to launch the filtering algorithm in order to avoid any misusage of the constraints. Typical examples of awakening conditions include the change of boundaries of variable domains involved into the relation or the assignment of one of its variables.

For example, SICStus Prolog `clp(FD)` allows the users to define new global constraints through an interface composed of two distinct parts:

1) `dispatch_global(Constraint, S0, S, Actions)` This predicate tells the solver that the predicate `Constraint` is a new global constraint. `Constraint` defines the interface of the constraint, while `Actions` define what actions to take. These actions include conditions when global inconsistency is proved (i.e., failing condition), when constraint entailment is proved (i.e., success), and partial satisfaction is obtained through variable binding or domain reduction. verb+S0, S+ are used to memorize previous and current states of the variable domains while applying the filtering algorithm ;
2) `fd_global(Constraint, State, Suspensions)` This predicate is used to post an instance of the constraint `Constraint`. `Suspensions` define the awakening conditions which include, in this implementation, variable binding, boundaries reduction or domain reduction. `State` contains the current state of the variables domain.

Using this mechanism, it is thus possible to implement various types of constraints, including those which are used to combine constraints with logical operators.

C. Syntax of the Constraint Language

The syntax of ITE constraint language is detailed in Fig. 1. The language contains several operators working on classical arithmetic constraints. It also introduces logical operators such as `cn`, `cd`, `cxd`, `=>`, `ite` which are typically useful to represent logical constraints. Note that the language introduces two version of constructive negation and constructive disjunction in order to distinguish the versions of these operators which implement stratified reasoning from the others. Both logical implication and if_then_else operators only use stratified reasoning as it is more useful (See Sec.IV for more details). This language is powerful as it allows us to perform various form of disjunctive reasoning.

III. Constructive Operators

Implementing meta-constraints such as disjunction, negation, implication, etc. requires to reason on the truth value of constraints and their possible (dis-)entailment by the current status of the CSP [3].

Definition 3: A constraint $C_k(X_j, \ldots, X_{j+r})$ is *entailed* by a CSP $(\mathcal{X}, \mathcal{C})$ iff, for any solution $v_j, \ldots, v_{j+r}$ of $(\mathcal{X}, \mathcal{C})$, the constraint $C_k(v_j, \ldots, v_{j+r})$ is satisfied.

Note that, by construction, all constraints of $\mathcal{C}$ are entailed by the CSP $(\mathcal{X}, \mathcal{C})$. In the above definition, C_k is not necessary part of $\mathcal{C}$.

Example 4: If all tuples of $D(X_1) = \{2, 3\}, D(X_2) = \{2, 3\}$ are solutions of a CSP $(\mathcal{X}, \mathcal{C})$ then the constraint $abs(X_1 - X_2)\#=<1$ is entailed by $(\mathcal{X}, \mathcal{C})$.

Proving constraint entailment or disentailment is as demanding as finding all solutions of a CSP. As solving CSP is NP_hard in the general case [10], relaxations of constraint entailment have been proposed.

A. Relaxations of Constraint Entailment

Entailment checking in practical settings is based on local filtering consistencies. Two partial entailment tests have been introduced in [10]: *domain-entailment* and *interval-entailment* which are based respectively on hyperarc-consistency and bound-consistency (See Sec.II). Another partial entailment test can be introduced here based on constraint refutation, namely Abs-entailment. *Abs-entailment* is a relaxation of entailment which exploits the filtering capabilities of the

$$
\begin{array}{lll}
CtrBody ::= & var \mid \texttt{true} \mid 1 \mid \texttt{false} \mid 0 & \\
\mid & var \texttt{ in } ConstantRange & \{ \text{ Domain constraint } \} \\
\mid & ExprArithm & \{ \text{ Arithmetical Expression with +,-,x,... } \} \\
\mid & CtrBody\ RelOp\ CtrBody & \{ \text{ Logical constraint } \} \\
\mid & CtrBody,\ CtrBody & \{ \text{ Constraint conjunction } \} \\
\mid & \texttt{cn}(CtrBody) \mid \texttt{cn}(CtrBody, Env) & \{ \text{ Statified constructive negation } \} \\
\mid & CtrBody\ \texttt{cd}\ CtrBody \mid \texttt{cd}(CtrBody, CtrBody, Env) & \{ \text{ Statified constructive disjunction } \} \\
\mid & CtrBody\ \texttt{cxd}\ CtrBody \mid \texttt{cxd}(CtrBody, CtrBody, Env) & \{ \text{ Statified constructive exclusive disjunction } \} \\
\mid & CtrBody \texttt{ => } CtrBody \mid \texttt{=>}(CtrBody, CtrBody, Env) & \{ \text{ Statified constructive implication } \} \\
\mid & \texttt{ite}(CtrBody, CtrBody, CtrBody, Env) & \{ \text{ Statified constructive if\_then\_else } \} \\
RelOp ::= & \texttt{\#=} \mid \texttt{\#<>} \mid \texttt{\#<} \mid \texttt{\#=<} \mid \texttt{\#>} \mid \texttt{\#>=} &
\end{array}
$$

Fig. 1. Syntax of ITE Constraint Language

entire CSP to try to refute the negation of the constraint. Using more formal presentation, we now describe these three relaxations of entailment which are available in ITE.

Definition 4: *(domain-entailment)*
Let $(\mathcal{X}, \mathcal{C})$ be a CSP having $(D_1, \ldots, D_n)$ as domains for its variables $X_1 \ldots, X_n$, then a constraint $C_k(X_j, \ldots, X_{j+r})$ is *domain-entailed* by $(\mathcal{X}, \mathcal{C})$ iff, for all domains D_i and all values $v \in D_i$, $C_k(v_j, \ldots, v_{j+i-1}, v, v_{j+i+1}, \ldots, v_{j+r})$ is satisfied.

Based on this definition, a constraint C_k can be entailed, but not necessary domain-entailed by a CSP $(\mathcal{X}, \mathcal{C})$. On the contrary, the opposite is true: any constraint C_k which is domain-entailed is also entailed by $(\mathcal{X}, \mathcal{C})$.

Example 5: If $X_1 \in \{2, 3, 4\}$, $X_2 \in \{2, 3, 4\}$ and $C = \{X_1 \#=< X_2\}$ then the constraint $X_1 \#=< X_2 + 1$ is entailed but not domain-entailed.

In this example, $X_1 \#=< X_2 + 1$ is entailed because all satisfying tuples of $X_1 \#=< X_2$, namely $\{(2,2), (2,3), (2,4), (3,3), (3,4), (4,4)\}$ are also solutions of $X_1 \#=< X_2 + 1$. However, this constraint is not domain-entailed because there exist pairs in $(\{2, 3, 4\}, \{2, 3, 4\})$ which do not satisfy $X_1 \#=< X_2 + 1$, e.g., the pair $(4, 2)$.

Domain-entailment is a *local property* as it requires only to examine the tuples of C_k with respect to the current domains of its variables. At the same time, proving domain-entailment is demanding as it requires to examine all satisfying tuples of C_k, which can be time-consuming when domains are huge. Another less-demanding relaxation is thus proposed in some solver implementations.

Definition 5: *(interval-entailment)*
A constraint $C_k(X_j, \ldots, X_{j+r})$ is *interval-entailed* by a CSP $(\mathcal{X}, \mathcal{C})$ iff, for all domains D_i and all values v in $min(D_i) \ldots max(D_i)$, $C_k(v_j, \ldots, v_{j+i-1}, v, v_{j+i+1}, \ldots, v_{j+r})$ is satisfied.

Obviously, if a constraint C_k is interval-entailed by a CSP $(\mathcal{X}, \mathcal{C})$, then C_k is also domain-entailed as $D_i \subseteq min(D_i)..max(D_i)$. But, the opposite is not true. Nevertheless, these two relaxed versions of entailment are quite similar and both are used in implementations of classical (non constructive) negation and disjunction. For instance, both are used in SICStus Prolog `clp(FD)` for implementing the reification operator. This operator is used to evaluate the truth value of any constraint, including some global constraints [2].

B. Abs-entailment

We introduce here another partial entailment test which is based on constraint refutation :

Definition 6: *(abs-entailment)*
A constraint C_k is *abs-entailed* by a CSP $(\mathcal{X}, \mathcal{C})$ iff filtering by *hyperarc-consistency* or *bound-consistency* the CSP $(\mathcal{X}, \mathcal{C} \wedge \neg C_k)$ yields to an inconsistency.

This definition is operational as it is based on concrete filtering properties that are available in most CSP solvers. As *hyperarc-consistency* and *bound-consistency* are sound relaxations of consistency, it is trivial to see that *abs-entailment* is a sound relaxation of entailment. Indeed, if $\mathcal{C} \wedge \neg C_k$ is inconsistent then $\mathcal{C} \implies C_k$ and then C_k is entailed.
Unlike *domain-entailment* or *interval-entailment*, *abs-entailment* is not a local property, restricted to the constraint C_k. It involves all the constraints of the CSP and requires the computation of $\neg C_k$. In addition, it requires the restoration of the computational state after the test of inconsistency of $(\mathcal{X}, \mathcal{C} \wedge \neg C_k)$. Nevertheless abs-entailment is powerful to deduce entailment and prune domains, more powerful than domain- and/or interval-entailment as shown in our experiments, given in Sec.V.

C. Constructive Disjunction

Abs-entailment can be used to implement constructive disjunction. By using the global constraint definition mechanism of a CSP solver, it becomes possible to introduce a new disjunctive constraint, i.e., $C_1\ cd\ C_2$, where one test whether

```
cd(C1, C2) :−
  term_variables(C1, L1), term_variables(C2, L2), %Get variables
  ord_union(L1,L2,L), add_dom(L,DOM), %awake when domains
       have changed
  clpfd:fd_global(cd_ctr(C1,C2,L),state,DOM).

clpfd:dispatch_global(cd_ctr(C1, C2, L),state, state, Actions) :−
  cd_solve(C1, C2, L, Actions).

% When there is no variable in the query (L == [])
cd_solve(C1, _C2, [], Actions) :− call(C1), !, Actions = [exit].
cd_solve(_C1, C2, [], Actions) :− call(C2), !, Actions = [exit].
cd_solve(_C1,_C2, [], Actions) :− !, Actions = [fail].

% if variables in the query (L \== [])
cd_solve(C1, C2, L, Actions) :−
  \+( (call(C1), assert_bounds(L)) ), !, % (C /\ C1)=>fail ?
  Actions = [exit, call(user:C2)].

cd_solve(C1, C2, L, Actions) :−
  \+( (call(C2), assert_bounds(L)) ), !, % (C /\ C2)=>fail ?
  retract(inb(_)),
  Actions = [exit, call(user:C1)].

cd_solve(_C1, _C2, L, Actions) :− % Else case
  union_bounds(L, Actions). % suspend and construct
```

Fig. 2. An implementation of constructive disjunction in `clp(FD)`

C_1 and C_2 are entailed or not. By adding C_1 (resp. C_2) to the constraint store and checking whether the resulting constraint system is inconsistent, one gets an easy abs-entailment test of $\neg C_1$ (resp. $\neg C_2$). Those tests can be performed each time the global constraint is awakened. Besides, while making these tests, one can register the domains status of each disjunct, after the filtering steps. Having filtered both $C \wedge C_1$ and $C \wedge C_2$ and stored domain information, it becomes possible to construct unified information for the variable domain of the constructive disjunction constraint. In order to illustrate this principle, an implementation of constructive disjunction is given in Figure 2. For readers who are familiar with Prolog, the implementation is straightforward. For the other readers, the interesting part of the implementation lies in the line with comment `%(C /\ C1)=>fail ?`. The corresponding Prolog code exploits the 'negation-as-failure' operator of Prolog to test whether $C \wedge C_1$ fails or not. By posting C_1 and checking the failure of the resulting computational state (by constraint propagation), we test if $\neg C_1$ is entailed. If the goal succeeds then the computation backtracks and undoes what has been deduced from adding C_1.

A typical example of using this implementation is given in the following example.

Example 6:
```
A,B,C in 1..5
Q: (A−B#=4)cd(B−A#=4), (A−C#=4)cd(C−A#=4)
A:  A,B,C in {1}\/{5}
```

This implementation raises the question of what to do when, during the inconsistency checks, other such tests have to be performed. This can happen, for example, in the presence of other `cd` constraints. The implementation given in Figure 2 will just recursively consider all such tests. This deep analysis may lead to a very precise result but, at the same time, will be computationally expensive. For this reason, we introduce in this paper stratified reasoning as a way to cope with this problem. Details are given in Sec.IV, but, let's examine first the other logical operators constructed using the very same principle.

D. Constructive Negation

Typical negation operators in Logic Programming implements negation-as-failure, which triggers failure when the negated goal succeeds and conversely. Although very useful in many contexts, this operator coincides with logical negation only when the negated goal is grounded, which means that all its variables are binded. Of course, when negating a constraint, there are several unbounded variables and then posting the negation of a constraint becomes problematic. Constructive negation in Constraint Logic Programming has been proposed to cope with this issue, using Clarke's completion process [7]. For CSP, only a few implementations have been considered. In ITE, we propose an implementation which is closed to constructive disjunction, by checking the inconsistency of the negated constraint. Using simple rewriting rules, negated compound constraints can be transformed into conjunction or disjunction of negated simpler constraints, as shown in Figure 3. Interestingly, the constructive negation operation is implemented using the global constraint interface of SICStus, so that, fine-grained domain pruning and suspension can be implemented.

E. Other Operators

Other logical operators based on similar principles can be implemented. In particular, exclusive disjunction, general constraint implication operator and conditional are easy to derive from the implementation of constructive disjunction. For exclusive disjunction `C1 cxd C2`, when one disjunct is shown to be inconsistent (e.g., `C1`), one can propagate the negation of this disjunct togather with the other disjunction (`call((cn(C1),C2))`), while this is not possible with non-exclusive disjunction. Note that conditional reasoning, i.e., if(C) then C_1 else C_2 can trivially be converted into an exclusive disjunction operator by using the formulae: $(C \wedge C_1) \vee (\neg C \wedge C_2)$. As CSP usually work on finite domains, the closed world hypothesis guarantees that there is no answer available where both C and $\neg C$ can be true. The following example illustrates the usage of `ite` when combined with other constraints.

Example 7:
```
Q:ite(I0#=<16, J2#=J0*I0, J2#=J0, ENV),
   J2#>8, J0#=2
A: J0 = 2, I0 in 5..16, J2 in 10..32
```

```
cn(C) :−
  term_variables(C, L), add_dom(L, DOM), %Get var, awake on
       dom
  clpfd:fd_global(cn_ctr(C,L),state,DOM).

clpfd:dispatch_global(cn_ctr(C,L),state,state,A) :−
  cn_solve(C, L, A).

% no variable in the query (eq. negation as failure)
cn_solve(true, _, Actions) :− !, Actions = [fail].
cn_solve(1, _, Actions) :− !, Actions = [fail].
cn_solve(false, _, Actions) :− !, Actions = [exit].
cn_solve(0, _, Actions) :− !, Actions = [exit].
cn_solve(C, [], Actions) :− call(C), !, Actions = [fail].
cn_solve(_C, [], Actions) :− !, Actions = [exit].

% Negation of simple constraints
cn_solve(in(V,R), _, Actions):− !, Actions = [exit,V in \ R].
cn_solve(X #> Y, _, Actions):− !, Actions = [exit, call(X #=< Y).
...
% Negation of constructive operators
cn_solve(cn(C),_, Actions):− !, Actions = [exit,call(C)].
cn_solve(C1 cd C2,_, Actions) :− !,
  Actions = [exit,call((cn(C1),cn(C2)))].
cn_solve((C1,C2),_, Actions) :− !,
  Actions = [exit, call(cd(cn(C1),cn(C2)))].
cn_solve(C1=>C2,_, Actions) :− !,
%Not(C1=>C2) <=> C1 /\ Not(C2)
  Actionss = [exit, call(cd(C1,cn(C2)))].
cn_solve(ite(C1,C2,C3, ENV), _, Actions) :− !,
%Not(C1/\C2),Not(C1)/\C3 <=> Not(C1) cd C2, C1 cd Not(C3)
  Actions = [exit, call((cn(C1) cd C2, C1 cd cn(C3)))].
...
```

Fig. 3. Excerpt of constructive negation implementation in `clp(FD)`

```
cd(C1, C2, ENV) :−
  term_variables(C1, L1), term_variables(C2, L2),
  ord_union(L1,L2, LI), get_reveil(ENV, Reveil),
  remove_var(LI, ENV, L), add_dom(L,DOM),
  clpfd:fd_global(cd_ctr3(C1,C2,L,ENV),s,[max(Reveil)|DOM).

clpfd:dispatch_global(cd_ctr3(C1,C2,L,ENV),s, s, Actions) :−
  cd_solve3(C1,C2, L, ENV, Actions).

% no variable in the query
cd_solve3(C1, _, [], _ENV, Actions) :− call(C1), !,
  Actions = [exit].
cd_solve3(_, C2, [], _ENV, Actions) :− call(C2), !,
  Actions = [exit].
cd_solve3(_,_, [], _ENV, Actions) :− !,
  Actions = [fail].
cd_solve3(_, _, _, ENV, Actions) :−
  test_k(ENV), !, Actions = []. % Test if K = 0 ?

cd_solve3(C1, C2, L, ENV, Actions) :−
  \+( (decr_k(ENV), call(C1), assert_bounds(L)) ), !,
  % K−1,C/\C1=>fail?
  Actions = [exit, call(C2)].

cd_solve3(C1, C2, L, ENV, Actions) :−
  \+( (decr_k(ENV), call(C2), assert_bounds(L)) ), !,
  % K−1,C/\C2=>fail?
  (retract(inb(_)) −> true),
  Actions = [exit, call(C1)].

cd_solve3(_C1, _C2, L, _ENV, Actions) :−
  union_bounds(L, Actions). % Construct and Suspend
```

Fig. 4. An implementation of stratified constructive disjunction in `clp(FD)`

Note that other `ite`operators can be used in branches of the operators, enabling cascade conditionals.

IV. Stratified Reasoning

In the previous section, we have proposed implementations of logical operators without paying attention to the computational costs of propagating constraints. Even though filtering by hyperarc-consistency and bound-consistency is time-polynomial in the worst case, the number of inconsistency checks performed can rapidly explode and leads to some unwanted combinatorial explosion.

In order to cope with this problem, we propose stratified reasoning. By setting up a user-defined parameter k to a positive integer (usually a small value), one can decrease k by one each time an inconsistency check is performed. When k reaches the value 0, then the inconsistency check is simply discarded, avoiding so to perform an uncontrolled and redundant exploration of all disjunctions. A straightforward implementation of this idea is shown in Figure 4. In this implementation, the variable `ENV` captures general information about the computation, including the value of k. While performing the inconsistency check (line with `%K−1,C/\C1 => fail?`), k is decreased. When $k = 0$, `Actions=[]` which corresponds to the suspension of any reasoning.

This way of handling disjunctive reasoning is correct and does not compromise the final result of CSP solving: it just reports to the final labeling stage the selection of a disjunction. Stratified reasoning is a good trade-off between search and inference, although coupled with a decision to which aspect is more important. The following example illustrates the benefice of stratified reasoning. By setting up different values of the user-defined parameter k, one gets distinct results with the same constraint system. Note that when the parameter is less than the number of `cd`operators (in the third case of the example), no deduction is obtained. We conjecture that k should always be setup to larger value than the number of disjunctive operators in the formulae but we do not have any formal proof of this. This conjecture is evaluated in the experimental evaluation section.

Example 8:

```
Q:init_env(E,7),cd(cd(X#=0,Y#=4,E),X#=9,E),
  cd(cd(Y#=9,Y#=6,E),Y#=7,E)
A:  X in{0}\/{9}, Y in(6..7)\/{9}

Q:init_env(E,6),cd(cd(X#=0,Y#=4,E),X#=9,E),
  cd(cd(Y#=9,Y#=6,E),Y#=7,E)
A:  X in inf..sup, Y in 6..9
```

```
Q:init_env(E,3),cd(cd(X#=0,Y#=4,E),X#=9,E),
  cd(cd(Y#=9,Y#=6,E),Y#=7,E)
A:  X in inf..sup, Y in inf..sup
```

V. Experimental Results

To confirm the effectiveness of the lightweight approach to constructive disjunction, we compare our ITE implementation to the corresponding standard methods contained in SICStus Prolog `clp(FD)`. We consider 28 representative expressions containing one or multiple constructive operators, including examples from [22] and [14]. The benchmark dataset and its execution scripts are available as part of the ITE distribution[2]. All experiments have been conducted with SICStus Prolog version 4.4.1 running on commodity hardware.

The evaluation goal is to identify scenarios that benefit from using ITE over the standard methods. For this goal, we compare the stratified and non-stratified constructive negation and disjunction operators from ITE against the `clp(FD)` operators `#\` and `#\/` as well as against the global constraint `smt` [2]. `smt` is a constraint that potentially allows stronger and faster propagation of reifiable constraints than the default propagators of `clp(FD)`, but not necessarily in all cases.

For operators with stratified reasoning, we experiment with different settings of the parameter k, which is adjusted to 0.9, 1, 1.25, 1.5, and 2 times the number of disjunctive operators in the formula, rounded to the next smallest integer. We tested multiplication factors larger than 2, but did not find better results, so we ignore them here. To evaluate the performance on the benchmark set, we both analyze the time to propagate the expressions and the resulting domain sizes. All expressions in the benchmark set can be resolved to finite domains, therefore we count propagations that do not return finite domains as incomplete propagations.

The experimental results, shown in figures 5 and 6, illustrate a trade-off between time for successful propagation and resulting domain size. Propagation via the built-in `clp(FD)` constraints is the quickest approach, although it does not necessarily propagates every expression to a finite domain. ITE successfully propagates all expressions in the benchmark set, although with time longer than `clp(FD)`, but shorter than `smt`. The results for the `smt` constraint are ambivalent as well. Propagation via `smt` is slower than native `clp(FD)` constraints, but more efficient in domain pruning, as it is also the case for ITE. However, the time of `smt` is much higher than of ITE and by choosing an appropriate value for k, ITE is more efficient in reducing the domain size.

Conclusively, for the trade-off between propagation time and resulting domain size, ITE is a beneficial choice, even without stratified reasoning. With stratified reasoning, choosing a value for k that is close to the number of disjunctive operators in the formula shows best performance.

[2]Available at https://github.com/ite4cp/ite

VI. Conclusion

In this paper, we have introduced a lightweight implementation of constructive disjunction and negation in Prolog-based CSP solvers. This implementation, called ITE, provides a variety of operators which are useful to explore logical combination of constraints. By exploiting abs-entailment, a powerful relaxation of entailment, and a global constraint definition mechanism, such an implementation is straightforward. As there is a risk of combinatorial explosion, our implementation provides stratified deduction, which is a user-parameterized technique to cope with the problem of disjunctive reasoning. Stratified deduction requires only to set up a single parameter, for which we provide a selection guideline.

Our experimental results show that ITE is competitive with available tools to deal with disjunctive reasoning, namely domain- and interval- entailment and the smt global constraint. As future work, we plan 1) to propose to automatically adjust the value of k for stratified reasoning through heuristic reasoning and 2) to explore the capability to infer more symbolic information from disjunctions than just pure domain information. The later proposition would require to propagate not only domains, but also deduced variable relations. This is a challenging problem as there does not exist any general recipe to perform the union of symbolic information in CSP solving.

References

[1] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Emmanuel Poder. New filtering for the cumulative constraint in the context of non-overlapping rectangles. *Annals of Operations Research*, 184(1):27–50, Apr 2011.

[2] Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, and Justin Pearson. On the reification of global constraints. *Constraints*, 18(1):1–6, 2013.

[3] B. Carlson, M. Carlsson, and D. Diaz. Entailment of Finite Domain Constraints. In *Proc. of 11th International Conference on Logic Programming*. MIT Press, 1994.

[4] M. Carlsson, G. Ottosson, and B. Carlson. An Open–Ended Finite Domain Constraint Solver. In *Prog. Lang., Impl., Logics, and Programs (PLILP)*, 1997.

[5] Stefano Di Alesio, Shiva Nejati, Lionel Briand, and Arnaud Gotlieb. Stress testing of task deadlines: A constraint programming approach. In *Int. Symposium on Soft. Reliability and Engineering (ISSRE'13), Research track*, Pasadena, CA, USA, Nov. 2013.

[6] Minh Binh Do and Subbarao Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into CSP. *Artif. Intell.*, 132(2):151–182, 2001.

[7] François Fages. Constructive negation by pruning. *J. Log. Program.*, 32(2):85–118, 1997.

[8] A. Gotlieb, T. Denmat, and B. Botella. Constraint-based test data generation in the presence of stack-directed pointers. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, Long Beach, CA, USA, Nov. 2005. 4 pages.

[9] A. Gotlieb and M. Petit. A uniform random test data generator for path testing. *The Journal of Systems and Software*, 83(12):2618–2626, Dec. 2010.

[10] Pascal Van Hentenryck, Vijav Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37:139–164, 1998. Also in CS-93-02 Brown–University 1993.

[11] Mikael Z. Lagerkvist and Christian Schulte. Propagator groups. In *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, pages 524–538, 2009.

[12] Christophe Lecoutre. *Constraint Networks: Techniques and Algorithms*. ISTE/Wiley. ISBN 978-1-84821-106-3, 2009.

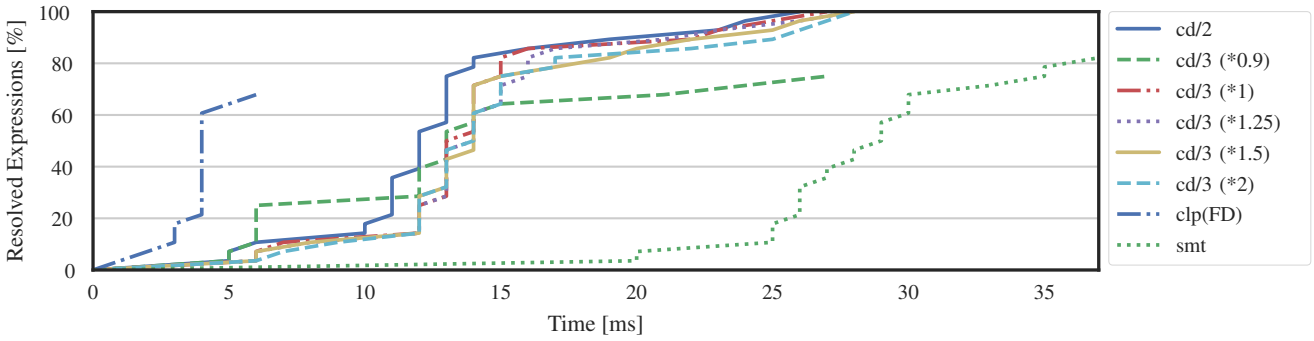

Fig. 5. Time to resolve expressions. An expression counts as resolved if it results in a finite domain, independent of the actual domain size. (see Fig. 6 for a comparison of the resulting domain sizes). `clp(FD)` is generally quick, but not able to resolve more than 30% of the expressions. The ITE variants cd/2 and cd/3 are able to resolve all expressions. An exception is cd/3 (*0.9) where the parameter k is smaller than the number of clauses in the expression.

	Opt.	+50%	+100%	$>100\%$	No Sol.
cd/2	100	0	0	0	0
cd/3 (*0.9)	50	7	7	11	25
cd/3 (*1)	100	0	0	0	0
cd/3 (*1.25)	100	0	0	0	0
cd/3 (*1.5)	100	0	0	0	0
cd/3 (*2)	100	0	0	0	0
clp(FD)	43	7	4	14	32
smt	54	7	4	18	18

Fig. 6. Domain size after propagation in relation to smallest domain returned by any of the propagators. *Opt.*: Resolved to smallest domain. *+50%/+100%*: Domains up to 50% respectively 100% larger. $>100\%$: More than twice the smallest domain. *No Sol.*: Not resolved to a finite domain.

[13] Kim Marriott and Peter J. Stuckey. *Programming with Constraints : an Introduction*. The MIT Press, 1998.

[14] Neil CA Moore and Patrick Prosser. The ultrametric constraint and its application to phylogenetics. *Journal of Artificial Intelligence Research*, 32:901–938, 2008.

[15] Morten Mossige, Arnaud Gotlieb, Helge Spieker, Hein Meling, and Mats Carlsson. Time-aware test case execution scheduling for cyber-physical systems. In *Proc. of Principles of Constraint Programming (CP'17)*, Aug. 2017.

[16] Bertrand Neveu, Gilles Chabert, and Gilles Trombettoni. When interval analysis helps inter-block backtracking. In *International Conference on Principles and Practice of Constraint Programming*, pages 390–405. Springer, 2006.

[17] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S., 2017.

[18] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[19] Christian Schulte and Peter J. Stuckey. Efficient constraint propagation engines. *Transactions on Programming Languages and Systems*, 31(1):2:1–2:43, December 2008.

[20] Gilles Trombettoni and Gilles Chabert. Constructive interval disjunction. In *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, pages 635–650, 2007.

[21] Ioannis Tsamardinos and Martha E Pollack. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence*, 151(1):43 – 89, 2003.

[22] Jörg Würtz and Tobias Müller. Constructive disjunction revisited. In *20th Annual German Conference on Artificial Intelligence (KI'96), Dresden, Germany*, pages 377–386, 1996.