1

High-Level Constraints over Finite Domains

M. Anton Ertl[1]
Andreas Krall

ABSTRACT Constraint logic programming languages that employ consistency techniques have been used to solve many combinatorial search problems. In solving such problems, the built-in constraints often do not suffice. Unfortunately, new constraints defined with lookahead and forward declarations are often inefficient. In this paper, we present an efficient high-level constraint mechanism. High-level constraints are ordinary predicates with an additional constraint declaration. They offer fine-grained control over the tradeoff between pruning power and execution time and achieve huge speedups over lookahead declarations.

1.1 Introduction

Many real-world problems, e.g. resource allocation and scheduling, can be solved using consistency techniques integrated with logic programming [VH89]. This integration consists of adding domain variables and constraints to Prolog. Domain variables are logic variables, that have an associated finite set of values, the (finite) domain. The domain explicitely represents the values that the variable can be instantiated with. Constraints are predicates that remove inconsistent values from the domains of their arguments.

For example, given the variables `X` with the domain $\{1, 2, \ldots 6\}$ and `Y` with the domain $\{4, 5, \ldots 9\}$, the constraint `X #> Y`[2] immediately reduces (prunes) the domains to $\{5, 6\}$ and $\{4, 5\}$ respectively. A constraint is usually activated again later, when the domain of an argument changes. In the example above, if `Y` is instantiated with 5 (i.e., its domain is reduced to $\{5\}$), the constraint is woken and instantiates `X` with 6.

For a network of constraints this behaviour results in local propagation over the domains. I.e., if the domain of a variable changes, the constraints on that variable are woken up. The constraints remove inconsistent values from the domains of their variables, waking up other constraints. This process continues until there are no more changes. To find solutions to the problem represented by the constraint network, this propagation is combined with a generator, e.g., a labeling predicate that instantiates each variable to some value of its domain.

There is one problem with this approach: the built-in constraints often do not suffice. They can be combined effectively in conjunctions, but not in disjunctions. Of course the usual Prolog approach to disjunctions can be used, i.e., creating a choicepoint. However, each choicepoint multiplies the work to be done, so this approach is often unacceptable. Even for conjunctions of constraints, there is sometimes a need for better pruning, which cannot

[1]Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, 1040 Wien, Austria; {anton,andi}@mips.complang.tuwien.ac.at

[2]In our constraint logic programming system, Aristo, `#>`, `#>=` etc. are the constraint versions of Prolog's `>`, `>=` etc. Declaratively, `#>` means the same as `>`.

Paper and BibTeX entry are available at http://www.complang.tuwien.ac.at/papers/. This paper was published in:
Manfred Meyer (ed.), Constraint Processing, Springer LNCS 923, pages 51--66
originally presented at: Constraint Processing, International Workshop at CSAM '93, pages 65--76

be met by the local propagation mechanism alone.

To alleviate this problem, Van Hentenryck introduced facilities for defining new constraints in the constraint logic programming language: forward- and lookahead-declarations. These mechanisms work by checking the consistency for every possible combination of values from the domains of the variables. This method is very slow, making user-defined constraints useless in many cases, where the reduction in search space does not amortize the higher cost of constraint propagation.

In this paper we present a new, more efficient and more flexible method for defining high-level constraints. The `constraint` declaration is introduced in Section 1.2. Section 1.3 shows that it does not change the declarative semantics and discusses related issues. Section 1.4 presents examples. The relation to forward- and lookahead-declarations is discussed in Section 1.5. Section 1.6 discusses efficiency improvements and shows how the mechanisms of high-level constraints can be used to define built-in constraints. Our implementation is described in Section 1.7 and Section 1.8 presents a few empirical results.

1.2 High-Level Constraints

We call our mechanism high-level constraints, because they are written in the constraint logic programming language itself. In contrast, low-level constraints are written in the implementation language of the system, e.g. C.

Syntactically, high-level constraints look just like ordinary predicates (they also have the same meaning, see Section 1.3.1), except that we have to add a constraint-declaration:

`:- constraint` C`=`$Head$ `trigger` $Goal_1$ `satisfied` $Goal_2$.

The predicate that is declared as constraint (the constraint predicate) is given by *Head*. $Goal_1$ specifies when the constraint is woken up; $Goal_2$ specifies when the constraint is satisfied[3]. Using metalogical goals for these purposes offers maximum flexibility. C is used in $Goal_1$ to reference the constraint.

How does a high-level constraint work? Whenever the constraint is woken, the constraint predicate is executed and all solutions of the predicate are collected. For each variable occuring in the arguments of the constraint, the union of its domains over all intermediate solutions is computed; then the domain of the variable is reduced to this union. This reduction of the domains is the entire effect of the constraint. Choicepoints and bindings made during the execution of the constraint have no effect outside. After this reduction, the satisfaction goal $Goal_2$ is executed. If it succeeds, the constraint is satisfied and need not be woken up any more. If it fails, the constraint is not yet satisfied and is just put back to sleep until it is woken up again. If the constraint goal does not have any solutions, the constraint fails.

The constraint is not executed in the usual environment: Constraints outside the high-level constraint (outer constraints) are not woken up in the high-level constraint. In other words, the constraint works as if outer constraints did not exist. During the collection of solutions,

[3] A constraint is satisfied, if it need not be woken up again.

floundering[4] constraints are ignored and the domains are used as they are.

The mechanism is illustrated by an example: the `max(X,Y,Z)` constraint holds if `Z` is the maximum of `X` and `Y`. It can be defined as follows:

```
:- constraint C=max(X,Y,Z)
     trigger trigger_minmax(X,C), trigger_minmax(Y,C),
             trigger_minmax(Z,C)
     satisfied one_var([X,Y,Z]).
max(X,Y,Z) :- X#>=Y, X=Z.
max(X,Y,Z) :- X#<Y,  Y=Z.
```

When `max(X,Y,Z)` is woken up and the variables have the domains

$$\mathtt{X} \in \{2,3,5\}, \quad \mathtt{Y} \in \{0,\ldots,4\}, \quad \mathtt{Z} \in \{2,4,6,8\}$$

`max(X,Y,Z)` produces two intermediate solutions:

$$\mathtt{X} = \mathtt{Z} = 2, \quad \mathtt{Y} \in \{0,\ldots,2\}$$

and

$$\mathtt{X} \in \{2,3\}, \quad \mathtt{Y} = \mathtt{Z} \in \{2,4\}$$

The unions are:

$$\mathtt{X} \in \{2,3\}, \quad \mathtt{Y} \in \{0,1,2,4\}, \quad \mathtt{Z} \in \{2,4\}$$

The effect of the activation of the constraint is to reduce the domains of the variables to these unions.

The `satisfied` goal in the example is `one_var(T)`, which succeeds if `T` contains at most one variable. This is a built-in, since it is frequently used (see Section 1.3.2).

Triggers work similar to Prolog II's `freeze/2`. They always succeed, and have the effect that the constraint is woken, when the variable's domain changes in a certain way.[5]

`trigger_minmax(V,C)` specifies that the constraint `C` is woken whenever the minimum or the maximum of the domain of `V` changes. It can be constructed from built-ins:

```
trigger_minmax(V,C) :- trigger_min(V,C), trigger_max(V,C).
```

We have four built-ins for specifying trigger conditions, corresponding to the wakeup mechanisms in our constraint logic programming system:

`trigger_ground(V,C)` Wake `C` up when `V` is instantiated (to an integer).

`trigger_min(V,C)` Wake `C` up whenever the minimum of the values in the domain changes, i.e., when the minimal value is removed from the domain.

[4] A constraint flounders if the computation ends before the constraint is satisfied. This can be avoided by sufficiently instantiating the variables.

[5] Note that a conjunction of triggers means that the constraint is woken if any of the trigger conditions is satisfied. The trigger syntax has been criticized as inelegant, but we did not find a more elegant syntax that offers the same (or better) expressive power and does not cause a big implementation effort.

`trigger_max(V,C)` Analogous to `trigger_min(V,C)`.

`trigger_size(V,C)` Wake `C` up whenever a value is removed from the domain.

The user can combine constraint declarations with coroutining declarations (e.g. NU-Prologs when-declarations). This can be used for delaying the first wakeup of the constraint until its arguments are instantiated sufficiently for achieving substantial pruning.

1.3 Properties and Restrictions

1.3.1 Declarative Semantics

The declarative semantics of a program with constraint declarations are exactly the same as without constraint declarations, if the following condition is met: The `satisfied` goal must not succeed unless all combinations of values of the domain variables are consistent with respect to the constraint (see Section 1.3.2).

High-level constraints are based on two mechanisms: triggering/activation and summarizing solutions. The triggering mechanism just changes the order of execution, which has no influence on the declarative semantics. A constraint may be activated several times, but only the last time is important for the declarative semantics, since any solution that passes that activation also passes the others.

To prove the equivalence of the declarative semantics of the summarizing process, we show that it does neither remove nor add any solutions. The summarizing process collects all solutions of the constraint predicate and processes them by taking the union, so the summary obviously contains all solutions. Due to the restriction on the satisfaction condition, the summary of a satisfied constraint contains only combinations of values that are consistent with respect to the constraint, i.e., it contains only solutions.

If the constraint is not yet satisfied, it flounders. We can think of a floundering constraint as qualified answer; it represents all consistent combinations of values of the variables. The floundering behaviour may be changed by the constraint declaration, but the answers represent the same solution set.

1.3.2 Satisfaction

The `satisfied` goal must only succeed if all combinations of the values in the domains are consistent. This is obviously the case when the constraint predicate succeeds and all arguments are ground. This condition is a bit too restrictive. A constraint is already satisfied, when there is at most one domain variable left [VH89] and all inconsistent values are removed from its domain, i.e. if there are no unsatisfied constraints within the high-level constraint. For some constraints it can be determined even earlier that there are no inconsistent combinations of values. E.g., `X#<Y` is satisfied when the largest value in the domain of `X` is smaller than the smallest value in the domain of `Y`.

```
:- constraint C = element(X,L,E)
     trigger trigger_size(X,C), trigger_size(E,C),
             trigger_listground(L,C)
     satisfied one_var([X,E|L]).
element(X,L,E):- element(X,L,E,1).

:- element(_,L,_,_) when L.
element(X,[E|Ls],E,X).
element(X,[_|Ls],E,N):- N1 is N+1, element(X,Ls,E,N1).

:- trigger_listground(L,_) when L.
trigger_listground([],_).
trigger_listground([V|Vs],C):- trigger_ground(V,C),
                               trigger_listground(Vs,C).
```

FIGURE 1.1. The `element/3` constraint

1.3.3 Pragmatics

Of course, maintaining the declarative semantics is not very useful if the program loops endlessly or flounders due to the change in procedural semantics. Fortunately, this is not a problem in practice: In typical applications, every domain variable is instantiated eventually. Therefore, every constraint is satisfied and there can be no floundering. Endless loops can usually be avoided by making liberal use of coroutining.

To ensure that there is no floundering, the `trigger` goal must specify enough wakeup conditions. As a minimum, the constraint should be woken when one of its arguments is instantiated (i.e., `trigger_ground`), otherwise it could miss a decisive instantiation and remain unsatisfied forever. Using other `trigger` goals causes more frequent wakeup[6] and earlier pruning, influencing performance. Which is best depends on the constraint and sometimes on the application.

1.4 Examples

The constraint `element(X,L,E)` holds if `E` is the X^{th} element of the list `L`. It is instrumental in solving many combinatorial search problems, e.g. the cutting stock problem of [VH89]. It can be defined as high-level constraint by just adding a constraint declaration to a declaratively programmed predicate (see Figure 1.1).

The `when`-declarations are coroutining declarations that delay the execution of the predicate until the specified arguments are instantiated.

The constraint `atmost(Nb,L,Val)` holds if at most `Nb` elements of `L` are equal to `Val`. This constraint is needed in solving a car-sequencing problem [DSVH88]. We could add a constraint declaration to the predicate described in [VHD91], but this would result in up

[6] `trigger_min/2` (`trigger_max`) must be used in conjunction with `trigger_ground/2` or `trigger_max/2` (`trigger_min/2`) to enforce triggering upon instantiation.

```
atmost(Nb,L,Val):- occurt(N,Val,L), N#=<Nb.

:- occurt(_,_,L) when L.
occurt(0,_,[]).
occurt(N+D,Val,[V|Vs]):-
      [D] in 0..1, occur1(D,V,Val), occurt(N,Val,Vs).

:- constraint C = occur1(D,V1,V2)
   trigger trigger_ground(D,C), trigger_ground(V1,C),
           trigger_ground(V2,C)
   satisfied one_var([D,V1,V2]).
occur1(1,V,V).
occur1(0,V1,V2):- V1 #/= V2.
```

FIGURE 1.2. The `atmost/3` constraint

to $2^{|L|}$ solutions per activation of the constraint, i.e., each activation could cost exponential time. Our implementation is natural and efficient (see Figure 1.2):

`occurt(Nb,Val,L)`[7] holds if exactly `Nb` elements of `L` are equal to `Val`. `occurt/3` creates one `occur1/3` constraint for each element, which checks the equality of one element with `Val`. `occurt/3` adds up the number of equalities, and `atmost(Nb,L,Val)` checks with the `#=</2` that they are less than or equal to `Nb`.

As soon as `Nb` variables in `L` become equal to `Val`, the sum reaches `Nb`, and the `#=</2` instantiates all other variables in the sum to 0. This in turn wakes the `occur1/3` constraints up, which then remove `Val` from domains of the rest of the variables via `#/=`.

1.5 Forward- and Lookahead-Declarations

This section describes the relation of our high-level constraint mechanism to forward- and lookahead-declarations. Basically, our high-level constraints can be seen as a generalization of lookahead declarations.

As an example, we will look at the constraint `disjunctive/4`. This constraint is used in scheduling applications.

A constraint declared with a lookahead declaration checks the consistency of every combination of values from the domains by running every combination through the predicate. Of course, this method is very time-consuming. In the example, if the variables `Si` and `Sj` have, e.g., 100 values in their domains, the constraint below would check 10 000 value combinations. Forward declarations can be mechanically transformed into high-level constraints with the same behaviour. E.g., the constraint

[7] A general-purpose `occur/3` can be defined by
`occur(N,Val,L):- occurt(Nt,Val,L), N#=Nt.`

```
:- lookahead disjunctive(domain, ground, domain, ground).
disjunctive(Si,Di,Sj,Dj) :- Sj #>= Si+Di.
disjunctive(Si,Di,Sj,Dj) :- Si #>= Sj+Dj.
```

is equivalent to

```
:- disjunctive(Si,Di,Sj,Dj) when Di and Dj.
:- constraint C=disjunctive(Si,Di,Sj,Dj)
     trigger trigger_size(Si,C), trigger_size(Sj,C)
     satisfied one_var([Si,Sj]).
disjunctive(Si,Di,Sj,Dj) :- labeling([Si,Sj]),
                            disjunctive1(Si,Di,Sj,Dj).

disjunctive1(Si,Di,Sj,Dj) :- Sj #>= Si+Di.
disjunctive1(Si,Di,Sj,Dj) :- Si #>= Sj+Dj.
```

`labeling/1` generates all combinations of values of the domain variables in its argument.

Forward declarations can be transformed in the same way, only the `when`-declaration is different; it allows only one variable:

```
:- disjunctive(Si,Di,Sj,Dj) when Di and Dj and (Si or Sj).
```

The high-level constraint version of the constraint is a bit slower than the lookahead declaration version, since the lookahead mechanism is more specialized. However, it is easy to make the constraint more efficient without losing pruning power: One variable can be left uninstantiated. This reduces the search process enormously, in the `disjunctive/4` example to less than 200 tries. This method works for all constraint predicates, unless they use built-ins that create a run-time error when presented with a variable.

Depending on the constraint, it may also be profitable to perform labeling after stating the subconstraints or to use domain-splitting instead of labeling, exploiting the pruning capabilities of the subconstraints.

We can do away with the labeling completely, if, for all alternatives, local propagation on the respective constraint networks is a satisfaction-complete[8] solver. This is the case for the `disjunctive/4` constraint[9], so searching is reduced to 2 tries.

For many constraints, it is also possible to use more restrictive triggering, resulting in fewer wakeups. E.g., in the case of `disjunctive/4`, `trigger_minmax/2` can be used without losing pruning power.

All these improvements and specialization of the satisfaction goal (Section 1.3.2) lead to the following `disjunctive/4`:

[8] In the CLP framework [JL87], a constraint solver is satisfaction-complete, if it can always decide whether a constraints network is satisfiable or not. Applied to finite domains this means that a satisfaction-complete solver always removes all values from the domains that are not consistent with the network of constraints. Partial lookahead constraints, forward-checking constraints and conjunctions of constraints are usually not satisfaction-complete.

[9] While the constraint `#>=/2` is not necessarily satisfaction-complete when applied to general linear terms, it is satisfaction-complete for simple cases like `V1 #>= V2+Const`.

```
:- disjunctive(Si,Di,Sj,Dj) when Di and Dj.
:- constraint C=disjunctive(Si,Di,Sj,Dj)
     trigger trigger_minmax(Si,C), trigger_minmax(Sj,C)
     satisfied dommin(Si,Sil), dommax(Si,Siu),
               dommin(Sj,Sjl), dommax(Sj,Sju),
               (Siu+Di=<Sjl; Sju+Dj=<Sil).
disjunctive(Si,Di,Sj,Dj) :- Sj #>= Si+Di.
disjunctive(Si,Di,Sj,Dj) :- Si #>= Sj+Dj.
```

All the above improvements have the same pruning power as lookahead-declarations. For many applications it is also profitable to trade pruning power for time spent in the constraint by performing less labeling or fewer activations.

The implementation of our high-level constraint mechanism is quite similar to the implementation of forward- and lookahead-declarations, and takes about the same effort.

1.6 Refinements

1.6.1 Outer Constraints

As described in Section 1.2, constraints that were invoked outside the high-level constraint are not activated in the high-level constraint. The reason for this is not a theoretical difficulty, but a practical consideration: Suppose that there is a network of high-level constraints that internally perform a lot of labeling. The labeling of one variable in the constraint would wake up other constraints which would label other variables and so on. Eventually this would result in a complete labeling of the network, with the associated exponential run-time behaviour, but without the usual benefit of labeling, namely, a solution.

On the other hand, it would be nice to apply outer constraints during the execution of high-level constraints to achieve better pruning. The problematic behaviour arises when outer constraints create choicepoints. The solution is to execute only constraints that cannot create choicepoints, i.e. low-level constraints.

1.6.2 Single Solutions

If a constraint predicate delivers only one solution (e.g., due to failure of all other alternatives), it no longer makes sense to apply the high-level constraint mechanism. The constraint predicate can be executed as normal predicate. It is then satisfied, unless it uses meta or extra-logical features. This avoids the overhead of solution collection and later wakeups. More importantly, it opens new avenues for pruning, since the subconstraints of such a high-level constraint are now visible outside.

built-in predicate	Behaviour
`dommin(X,MinX)`	Unifies `MinX` with the smallest value in the domain of `X`
`dommax(X,MaxX)`	Unifies `MaxX` with the largest value in the domain of `X`
`domlist(X,L)`	Unifies `L` with the list of values in the domain of `X`
`not_equal_const(X,Y)`	`Y` must be an integer; remove `Y` from the domain of `X`
`greater_equal_const(X,Y)`	`Y` must be an integer; remove values <`Y` from the domain of `X`
`less_equal_const(X,Y)`	`Y` must be an integer; remove values >`Y` from the domain of `X`

TABLE 1.1. These built-in predicates can be used to build any constraint

1.6.3 Built-in Constraints

Our mechanism for defining high-level constraints can also be used by the language implementor for defining built-in constraints. E.g., `#=</2`[10] can be defined by:

```
:- builtin_constraint C = X#=<Y
     trigger trigger_min(X,C), trigger_ground(X,C),
             trigger_max(Y,C), trigger_ground(Y,C),
     satisfied dommax(X,MaxX), dommin(Y,MinY), MaxX=<MinY.
X #=< Y :- dommin(X, MinX), greater_equal_const(Y, MinX),
           dommax(Y, MaxY), less_equal_const(X, MaxY).
```

`dommin/2` and `dommax/2` are metalogical predicates for getting the minimum/maximum value in the domain of the variable. Integers are treated as domain variables with one value in the domain. `greater_equal_const/2` and `less_equal_const/2` are built-in predicates (i.e., not constraints) for updating the minimum/maximum of the variable's domain. A few such predicates (see Table 1.1) make it possible to define all constraints with the high-level constraint mechanism.

For such constraints the important feature of high-level constraints is the wakeup mechanism. Most of them have only one solution, so the solution collection mechanism is not necessary and would constitute unnecessary overhead. The constraint predicate can simply be executed. Note that, since the constraint predicate uses metalogical goals, the constraint is not satisfied after executing the predicate once, even though it has only one solution. Instead, it is satisfied only when the `satisfied` goal becomes true. Declaring the constraint as `builtin_constraint` (instead of `constraint`) indicates that this different treatment should be applied.

The benefits of using high-level constraints for defining built-ins are lower development costs. E.g., the low-level version of `#=</2` constraint defined above takes 45 lines of C code in the Aristo system. In addition, if all low-level constraints are eliminated, the mechanism for handling them becomes unnecessary. The cost is of course lower performance. However, by using optimizing compiler technology [Tay90, KB92] this cost can be eliminated.

[10] This version cannot work with linear terms, only with plain variables. The general version requires more code.

Solutions	low-level	forward	high-level	lookahead	high-level	choice
one	3.53ms	4.90ms	3.93ms	18.24ms	14.65ms	10.80ms
all	5.92ms	9.75ms	7.19ms	47.00ms	38.90ms	16.49ms

TABLE 1.2. Five houses timings

1.7 Implementation

When a high-level constraint is called, the system creates a *frozen constraint*, a data structure similar to a suspension [Car87] that contains the constraint goal and the `satisfied` goal. Then the `trigger` goal is called; the variable `C`, used as the second argument to the trigger built-ins, contains the frozen constraint. A trigger built-in inserts the frozen constraint into the list of constraints that are woken up when the specified attribute of the variable changes. E.g., `trigger_min(V,C)` inserts the frozen constraint `C` into the min-list of `V`.

What happens when the constraint is woken up depends on its type. For a constraint declared with `constraint`, room sufficient for the domains of the variables in the arguments of the constraint is reserved. In this space the resulting domains are constructed. They are initialized to empty. Then a procedure similar to `findall/3` is performed: If there is no solution, the constraint fails and backtracking is performed. If there is a solution, the values in the domains of the variables in the arguments are added to the appropriate result domains. Then backtracking is initiated. If there is another solution, then the process is repeated. Finally, the constraint goal will fail. If there was only one solution, the constraint is marked as satisfied and the constraint goal is meta-called (Section 1.6.2). Otherwise, the variables in the arguments of the constraint are reduced to the result domains; and the `satisfied` goal is called. If it succeeds, the constraint is marked as satisfied, otherwise as frozen (i.e., it might be woken up again).

The processing of a `builtin_constraint` is much simpler: The constraint goal is simply meta-called. Then the `satisfied` goal is called and processed as described above.

The restriction of not executing outer high-level constraints is implemented as follows: The system counts the depth of constraint processing; every frozen constraint has a field containing the depth where the constraint was called. A high-level constaint is only woken up, if the current level is the same as the level at the call of the constraint.

This implementation adds a few restrictions: The `trigger` and `satisfied` goals must not have any declarative meaning, i.e., they must not bind arguments of the constraint or add any constraints; they also must not create (permanent) choicepoints. The `trigger` goal must always succeed. These restrictions can be enforced at run-time by checking the last choicepoint and the entries on the trail.

1.8 Results

We implemented the high-level constraint mechanism described here. The times were measured using the Aristo system, a WAM-emulator based constraint logic programming system.

```
?- [A,B,C] in 1..N, max(A,B,C).
```

domain size N	hlc	lookahead
20	0.87ms	2360ms
50	1.12ms	36500ms
100	1.55ms	292000ms

TABLE 1.3. `max/3` timings (one activation)

We compared four approaches: using a low-level constraint, our high-level constraint mechanism, forward checking, and using disjunctions as choices. We used the five houses puzzle [VH89] as benchmark. The solution involves a constraint `plusorminus/3`, which we implemented with all approaches. We have measured two high-level constraint versions: one with the same pruning power as forward checking, one with the same pruning power as lookahead. The times for finding the solution on a DecStation 5000/150 (50/100Mhz R4000, 46 SPECInt) are shown in Table 1.2. The *all* times include the time for proving that the puzzle has only one solution.

We also compared the performance of one activation of the `max/3` of Section 1.2 with a full labeling (lookahead style) version of `max/3` (see Table 1.3, timings performed on a DecStation 5000/125 (25MHz R3000, 16 SPECInt)). Both versions have the same pruning power. As expected, the lookahead version becomes slower with the cube of the domain size. The high-level constraint version is between 2700 and 188000 times faster. Domain sizes in this range are not unusual for `max/3`. The significance of this example is not the enormous speedups; instead, it shows that lookahead declarations are often too slow in practice, while high-level constraints are usable.

1.9 Related Work

Section 1.5 compares our high-level constraints to forward- and lookahead-declarations [VH89].

The cardinality operator [VHD91] can express disjunction and negation of constraints. It is based on constraint entailment (implication). In principle it can be applied to any domain (e.g. rationals). In contrast, our mechanism is restricted to finite domains, but it achieves better pruning. E.g., a cardinality operator version of `max/3` achieves no pruning for the example in Section 1.2, since it cannot show negative entailment of one of the two branches of `max/3`. The `atmost/3` example gives an idea how the cardinality operator can be emulated with high-level constraints.

Constraint simplification rules [Frü92] replace or augment (combinations of) constraints with simpler constraints. They can simplify delayed user-defined predicates like `max/3`, but they achieve no pruning for the example in Section 1.2. Simplification rules are practical for stating more global relations and are complementary to high-level constraints.

Echidna [SH92] allows disjunctions of inequalities over real domains. The method employed for disjunctions can be seen as a specialization of our method for summarizing alternative intermediate solutions.

Generalised propagation [LPW92] extends the propagation mechanism from finite domains to arbitrary domains. A propagation step for a goal results in an approximation of the solution set that is as close as can be represented in the domain. [LPW92] does not say much about how to compute these approximations and when to perform propagation steps (when to wake up constraints). For finite domains, lookahead declarations are a specialization of generalized propagation. High-level constraints are a bit more general, as they can also express partial lookahead, i.e., an approximation that is not as close as possible.

[VHSD91] presents many ideas, among them constructive disjunction of constraints, which is further explored in [JS93]. As described in the latter paper, constructive disjunction is a combination of our summarizing process (called *global lookahead reduction* in [JS93]) with a specialization of the cardinality operator. All of these cardinality operator features are covered by our single solution refinement (see Section 1.6.2), except positive reduction which causes just early satisfaction of the constraint, but does not add to the pruning power. Surprisingly for papers that closely related, the discussion in [JS93] is quite disjoint with that in the present paper. E.g., there is no discussion of wakeup conditions.

The Nicolog system [Sid93] uses a functional intermediate representation (the projection language) for constraint networks. Among other things, new constraints can be defined in this language. However, this language manipulates domains explicitly and is therefore lower-level than the logic programming approach used for high-level constraints defined with `constraint` declarations. On the other hand, it is higher-level than building constraints with `builtin_constraint` and the built-in predicates for manipulating domains.

[VHSD91] presents a powerful built-in constraint, similar to Nicologs projection language, which is described in more detail in [DC93]. The same comments apply as for Nicolog.

There is an even lower level than `builtin_constraint`, which relies on domain variables and a constraint wakeup mechanism: [CFS93] implements a finite domain constraint solver with coroutining and backtrackable assignment. Metastructures [Neu90] support user-enhanced unification for building constraint solvers [Hol90] or coroutining.

Coroutining delays the execution of goals until they are sufficiently instantiated [CMG82, Nai86]. Like constraint declarations, coroutining declarations do not change the declarative meaning of a program. The implementation of high-level constraints is closely related to the implementation of coroutining. However, they are different mechanisms for different purposes.

1.10 Further Work

[LPW92] inspires the idea that a high-level constraint might not only reduce the domains, but might also add constraints that are satisfied (entailed) for all intermediate solutions. Since checking the entailment of every possible constraint is too much overhead, only a few constraints that are likely to be entailed should be checked. These constraints could be specified by the programmer or determined in a training run.

Specifying trigger and satisfaction conditions is too much work and error-prone. It may be possible to generate good conditions automatically, leaving only fine-tuning to the programmer.

Another interesting topic is negative high-level constraints. Currently, we can use De-Morgan's Law to push the negation down and then use negative versions of the built-in constraints. Apart from automating this transformation, there's also the negative forward checking inference rule [VH89].

1.11 Conclusion

We have presented a mechanism for defining high-level constraints over finite domains. It is achieved by mechanisms for waking constraints and for collecting and summarizing the solutions of a constraint predicate. Syntactically, high-level constraints are normal predicates with an additional constraint declaration.

High-level constraints allow a very fine-grained control over the choice between pruning power and execution time cost. The declarative meaning of the program is the same with and without constraint declarations. Our mechanism subsumes the domain- and forward-declaration mechanism of [VH89] and is as easy to implement. As free bonus, the builtin constraints can be defined with this mechanism.

Acknowledgements: We are grateful to Franz Puntigam, Ulrich Neumerkel, Gregory Sidebottom, Konrad Schwarz, Thomas Graf, Bruno De Backer and Hendrick Lock for commenting on earlier versions of this paper, and to DMS Decision Management Systems GmbH for making Aristo available to us.

1.12 References

[Car87] Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In *Fourth International Conference on Logic Programming (ICLP-4)*, pages 40–58. MIT Press, 1987.

[CFS93] Phillippe Codognet, François Fages, and Thierry Sola. A metalevel compiler of CLP(FD) and its combination with intelligent backtracking. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 437–456. MIT Press, 1993.

[CMG82] K. L. Clark, F. G. McCabe, and S. Gregory. IC-Prolog language features. In K. L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, London, 1982.

[DC93] Daniel Diaz and Phillippe Codognet. A minimal extension of the WAM for `clp(fd)`. In *International Conference on Logic Programming (ICLP)*, pages 774–790, 1993.

[DSVH88] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving the Car Sequencing Problem in Constraint Logic Programming. In *European Conference on Artificial Intelligence (ECAI-88)*, München, 1988.

[Frü92] Thom Frühwirth. Constraint simplification rules. Technical Report ECRC-92-18?, ECRC, 1992.

[Hol90] Christian Holzbaur. *Implementation of Constraint Based Inference Mechanisms through Extended Unification*. PhD thesis, Technische Universität Wien, 1990.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Fourteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 111–119, München, 1987.

[JS93] Jean Jourdan and Thierry Sola. The versatility of handling disjunctions as constraints. In *Programming Language Implementation and Logic Programming (PLILP)*, pages 60–74, 1993.

[KB92] Andreas Krall and Thomas Berger. Fast Prolog with a VAM_{1p} based Prolog compiler. In *Programming Language Implementation and Logic Programming (PLILP '92)*, pages 245–259. Springer LNCS 631, 1992.

[LPW92] Thierry Le Provost and Mark Wallace. Domain independent propagation. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 1004–1011, ICOT, Japan, 1992. Association for Computing Machinery.

[Nai86] Lee Naish. *Negation and Control in Prolog*. Springer LNCS 238, 1986.

[Neu90] Ulrich Neumerkel. Extensible unification by metastructures. In *Meta-90*, Leuven, 1990.

[SH92] Gregory Sidebottom and William S. Havens. Hierarchical arc consistency for disjoint real intervals in constraint logic programming. *Computational Intelligence*, 8(4):601–623, 1992.

[Sid93] Greg Sidebottom. Compiling constraint logic programming using interval computations and branching constructs. Technical report, Simon Fraser University, 1993.

[Tay90] Andrew Taylor. LIPS on a MIPS. In *Seventh International Conference on Logic Programming (ICLP-7)*, pages 174–185. MIT Press, 1990.

[VH89] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming Series. MIT Press, Cambridge, Massachusetts, 1989.

[VHD91] Pascal Van Hentenryck and Yves Deville. The cardinality operator: A new logical connective for constraint logic programming. In *Eighth International Conference on Logic Programming (ICLP-8)*, pages 745–759. MIT Press, 1991.

[VHSD91] Pascal Van Hentenryck, Vijay Saraswat, and Yves Deville. Constraint processing in `cc(FD)`. Ftp from parcftp.xerox.com, file pub/ccp/ccfd/pldi-5.ps, 1991.