Università degli Studi di Bologna

DEIS

A Meta Constraint Logic Programming Scheme

Evelina Lamma   Paola Mello   Michela Milano

October 10, 1995

DEIS Technical Report no. DEIS-LIA-95-005   *LIA Series no. 5*

A Meta Constraint Logic Programming Scheme

Evelina Lamma [1] Paola Mello [2] Michela Milano [1]

[1] *DEIS, Università di Bologna*
Viale Risorgimento 2, 40136 Bologna, Italy
Tel: +39 51 6443033, Fax: +39 51 6443073
{elamma,mmilano}@deis.unibo.it
[2] *Dipartimento di Informatica, Università di Bari*
Via Orabona 4, 70126 Bari, Italy
pmello@deis.unibo.it

October 10, 1995

Abstract. We present a Meta Constraint Logic Programming (MCLP) general scheme. The idea behind the work concerns the possibility of building meta CLP architectures by adding CLP solvers as meta level reasoners on the constraints of the underlying object systems. In this way, we are able not only to increase the expressive power of CLP, but also to implement several constraint satisfaction techniques like arc, path and in general, k-consistency by using lower degrees of consistency algorithms in order to obtain higher degrees of consistency.

The purpose of the work is to provide a general scheme which can be specialized in those domains and applications where the expressive power of CLP does not suffice. A distinguishing feature of the architectural scheme concerns its operational semantics and, in particular, the linking rules between different levels.

In this paper, we propose, as an example, two different specializations of a Meta CLP on finite domains (FD) architecture: the first concerns the possibility of embedding the capability of performing qualitative reasoning in a CLP framework, while the second concerns a multi-level architecture for obtaining different degrees of consistency by using weaker algorithms.

Keywords: *Constraint Logic Programming, Meta-Level Reasoning, Qualitative Reasoning, Consistency Algorithms*

Contents

1 Introduction

Constraint Logic Programming[1] (CLP) is a class of programming languages based on the Logic Programming paradigm [21, 25]. CLP combines the declarative semantics of Logic Programming with the efficiency of constraint solving, which replaces the unification mechanism. In this way, the applicability of Logic Programming is substantially increased.

A second feature of CLP concerns the possibility of specializing the CLP general scheme [18] in a specific domain. Working in a pre-defined domain ensures that the programmer uses concepts that are closer to the application domain. Associated with each domain, there are primitive operations and relations (constraints) on the domain itself.

These features have lead CLP to receive increased attention in recent years from both a theoretical and a practical point of view. Various constraint systems have been developed, most notably, CHIP [5, 11], CLP($\Re$) [20], Prolog III [7], ECL^{i}PS^e [13] and ILOG [31]. These tools contain constraint solvers in specified domains. The most common constraint solvers involve:

- rational and real numbers based mainly on simplex and gaussian elimination;
- boolean algebra based on variable elimination on a graph representation;
- finite domains based on local consistency techniques and bound propagation.

These constraint systems have been used for solving several real-life applications such as graph coloring, planning, scheduling, sequencing and assignment problems to name a few (see for instance [9, 10, 12, 32]).

However, there are some problems and applications where the expressive power of CLP does not suffice. For example, qualitative reasoning needs a tool which is able to reason and infer information on constraints such as the determination of the tighter constraint which holds between two variables. Therefore, we need a programming paradigm which is able to reason on constraints.

Constraint Logic Programming has very flexible and efficient constraint handling mechanisms and, therefore, can be easily extended in order to cope with this kind of problems.

In this paper, we show a general Meta Constraint Logic Programming scheme which can be specialized on particular domains and applications where the expressive power of CLP does not suffice.

A meta-system [27] is a computational system which has as its domain another computational system, called its object-system. Therefore, a meta-system is a system reasoning and acting upon another computational system by representing the object-system in its data.

[1] For a survey on CLP see [19].

In the field of Logic Programming (LP) [25] several efforts have been performed in order to develop meta-systems dealing with the LP provability relation [1, 4]. Constraint Logic Programming, while maintaining the features of LP and the possibility of building LP meta-systems, extends LP with constraint-solving mechanisms. In this paper, we focus on a meta system dealing with the extension of LP in terms of constraint solving. In our meta-architecture each meta-system reasons on constraints of the underlying object-system and applies propagation mechanisms to the constraints of the underlying levels.

A distinguishing feature of the architectural scheme concerns its operational semantics and, in particular, the linking rules between different levels. These rules link the constraint stores of different levels in order to exchange information and to propagate constraints.

The most attractive feature of the architecture concerns the very easy implementation of specializations of the system. We can build homogeneous systems by adding several equal constraint solvers or we can build heterogeneous systems by adding different constraint solvers each reasoning on the constraints of the underlying system.

We propose two different specializations of a meta CLP(FD) architecture: the first concerns the possibility of embedding in a CLP framework the capability of performing qualitative reasoning. In this specialization, the object level is a constraint solver on finite domain of integers, while the meta level is a constraint solver on finite domain of sets of constraints.

The second specialization concerns a multi-level architecture for reaching different degrees of consistency by using weaker algorithms. The different levels are CLP(FD) solvers adopting a k-consistency propagation algorithm [14]. By adding levels, we are able to reach higher degrees of consistency by using algorithms with a lower degree of consistency.

The paper is organized as follows: Section 2 is devoted to the Meta Constraint Logic Programming general scheme. Section 3 describes its operational semantics in terms of the object-level system, the meta-level system and the linking rules between them. In Section 4, we present a specialization concerning the consistency algorithms, while in Section 5, we present a specialization of the general scheme concerning qualitative reasoning. Conclusions and future works follow.

2 Meta CLP General Scheme

Meta-systems have been explored widely in the fields of Artificial Intelligence and computer science. A meta-system [27] is a computational system that has as its domain another computational system, called its object-system. Therefore, a meta-system is a system reasoning and acting upon another computational system by representing the object-system in its data.

In the field of Logic Programming (LP) [25] several efforts have been performed in order to develop meta-systems dealing with the LP provability relation [1, 4]. The resulting systems are very easy to maintain, extend and modify.

Constraint Logic Programming, while maintaining the features of LP, extends LP with constraint solving mechanisms. Therefore, in CLP we have the possibility of building meta-systems as well as in LP. In fact, CLP meta-systems maintain the advantages of the LP meta-systems.

We are interested in the CLP meta-systems dealing with the extension of LP in terms of constraint solving. In our meta-architecture, in fact, each meta-system reasons on constraints of the underlying object-system and applies propagation mechanisms on constraints of the underlying levels. In the following we present how this architecture can be built.

Each CLP language can be seen as an instance of a general CLP(X) scheme where each instance can be specified by instantiating the parameter X (for more details see [19]). This parameter X represents a 4-tuple $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$ where Σ is a signature, $\mathcal{D}$ is a Σ-structure, $\mathcal{L}$ is a class of Σ-formulas and $\mathcal{T}$ is a first-order Σ-theory.

The signature Σ represents the predefined predicates and function symbols, $\mathcal{D}$ is the domain structure, $\mathcal{L}$ is a class of constraints which can be expressed in the domain and $\mathcal{T}$ is the axiomatization of some properties of $\mathcal{D}$. Each CLP language can be defined by specifying all the 4-tuple components. Each CLP system is determined by the constraint domain $(\mathcal{D}, \mathcal{L})$ and by the operational semantics.

We propose a meta CLP general scheme composed by a number $n \geq 1$ of modules. Each module presents the above-mentioned structure: $(\Sigma, \mathcal{D}, \mathcal{L}, \mathcal{T})$. The most important characteristic of the scheme is that each level reasons on the constraints of the underlying level. Therefore, if $(\Sigma_n, \mathcal{D}_n, \mathcal{L}_n, \mathcal{T}_n)$ is the domain of a level n, the domain of the level $n+1$ is determined as a general function of the domain and constraints of the level n:

$$\mathrm{D}_{n+1} = f(\mathrm{D}_n, \mathcal{L}_n)$$

The signature of the $n+1$ level contains an implicit or explicit representation of the constraints in $\mathcal{L}_n$, the operations on the constraints of $\mathcal{L}_n$ and the constraints (meta-constraints) between them. With implicit we mean that we represent the constraints as symbols while an explicit representation can be handled only if we work on finite domains and it is composed of all the n-tuples of values allowed by n-ary constraints[2].

The representation of the meta-signature should be defined according to the application we have to solve. In Sections 4 and 5 we will specialize this general framework with two different applications.

[2]Usually, we work with binary constraints because each n-ary relation between variables can be represented in terms of binary constraints.

In the first specialization, we will represent the meta-signature symbols as n-tuples allowed by the constraints of the underlying system and, thus, providing an explicit representation. The function f of $\mathrm{D}_{n+1} = f(\mathrm{D}_n, \mathcal{L}_n)$ provides the representation of D_{n+1} as the powerset of constraints in $\mathcal{L}_n$.

In the second specialization, we will treat qualitative reasoning. The meta-signature symbol representation is given in terms of constraint symbols of the underlying system and, thus, providing an implicit representation. In this case, the function f provides the representation of D_{n+1} as the cartesian product of D_n.

The choice between implicit or explicit notation is domain dependent.

In each specialization, for each constraint linking two (or in general n) variables, we have a meta-variable which represents (explicitly or implicitly) this relation. The meta-system then propagates meta-constraints and performs meta-operations on constraints of the underlying system. Finally, the meta-system propagates the results in the underlying object system.

3 Operational Semantics

As already stated before, Constraint Logic Programming has been defined as a general scheme which can be specialized by defining the constraint domain and the operational semantics. The general constraint domain, which can be specialized for different applications, has been defined in Section 2. In this Section, we present a general operational semantics scheme suitable for each CLP language.

We will then extend this semantics in order to deal with the meta CLP architecture. Note that the general operational semantics for each module (which is a constraint solver) is the same as the semantics of CLP. Moreover, we have to define the operational semantics for the interactions of the two systems (in the following sections we will call two related components the object and the meta system).

In order to define a meta CLP general scheme, we have not only to present a general operational semantics for each architecture component, which is basically a CLP solver, but also we have to define the linking rules between different components, i.e., between each component and its underlying system.

Therefore, in the following subsections we will present the operational semantics of a single component of the meta CLP scheme and the linking rules between two components.

3.1 Operational Semantics of a Single Component

A single component of the meta CLP general scheme is a constraint system. The operational semantics of a constraint system [19] can be defined in terms of transition rules which allow the system to switch from different states, i.e. the system performs the so called "top-down execution"[3].

States are represented by tuples $\langle A, C, S\rangle$ where A is a multiset of atoms and constraints, C is the constraint store of active constraints, i.e., constraints that are *awake*, and S is the constraint store of passive constraints, i.e., constraints that are *asleep*. The failure of the system is represented by a state called *fail*. The transition system is very general but can be parameterized by defining the set of active and passive constraints and two predicates tailored on the particular domain: *consistent* and *infer*.

In the following, we report the general scheme of the operational semantics of a CLP system, taken from [19]. The first transition is denoted by $\rightarrow_r$ where r stands for resolution:

$$\langle A \cup a, C, S\rangle \rightarrow_r \langle A \cup B, C, S \cup (a = h)\rangle$$

if a is selected by the computation rule, a is an atom, and $h \leftarrow B$ is a rule of the program P, renamed to new variables, and h and a have the same predicate symbol.

It is worth noting that in this transition we do not need to use a substitution because the term $a = h$ is stored in the constraint store S and will be processed in the following steps. Unification, in fact, can be seen as a constraint solving process.

The $\rightarrow_r$ can lead to a failure if h and a have not the same predicate symbol for each clause of the program P:

$$\langle A \cup a, C, S\rangle \rightarrow_r fail.$$

The second transition is denoted by $\rightarrow_c$, where c stands for constraint, and introduces constraints into the constraint store:

$$\langle A \cup c, C, S\rangle \rightarrow_c \langle A, C, S \cup c\rangle$$

where c is a constraint selected by the computation rule.

The third transition rule, denoted by $\rightarrow_i$ where i stands for infer, infers new active constraints and modifies (generally simplifies) the passive constraints starting from the current collections of constraints C and S with the predicate $(C', S') = infer(C, S)$:

$$\langle A, C, S\rangle \rightarrow_i \langle A, C', S'\rangle.$$

The last transition rule, denoted by $\rightarrow_s$, where s stands for satisfiability, tests whether the active constraints are consistent with the predicate $consistent(C)$. In complete constraint systems[4] the predicate $consistent(C)$ holds iff $\mathcal{D} \models \tilde{\exists}C$[5]. Therefore, if $consistent(C)$ the transition is

[3] There are other definitions for Operational Semantics of CLP, see for example [33].

[4] Some CLP systems, however, have an incomplete consistency check.

[5] The formula $\tilde{\exists}C$ denotes the existential closure of C.

$$\langle A, C, S\rangle \rightarrow_s \langle A, C, S\rangle$$

otherwise, the transition is

$$\langle A, C, S\rangle \rightarrow_s fail.$$

A derivation is a sequence of transitions $\langle A_1, C_1, S_1\rangle \rightarrow \ldots \rightarrow \langle A_i, C_i, S_i\rangle \rightarrow \ldots$ where $\rightarrow$s are one of the above-mentioned transitions, the starting state is $\langle Goal, \emptyset, \emptyset\rangle$ and the final state is a state which cannot be rewritten further. This final state is *successful* if has the form $\langle \emptyset, C, S\rangle$, otherwise it is *failed*.

3.2 Interactions between Two Components

The most interesting feature of the system concerns the interaction between two levels. A change in the meta-level should be reflected in the object-level and viceversa. Therefore, some *linking rules* between the two systems should be defined.

The active meta constraint store contains a representation of the passive object constraint store. The meta level, in fact, reasons on the constraints (passive constraints) of the underlying system. For example, if the passive object constraint store contains the constraint $X \leq Y$, the meta active constraint store should contain its representation. A possible representation can be given in terms of a meta variable XY whose domain contains the constraint symbols linking the object level variables X and Y. The constraint stores of the two systems, i.e., the object- and meta-active and passive constraint stores (C_{obj}, C_m, S_{obj}, S_m respectively), can be linked as depicted in the Figure 1.

Consider the linking rule from the meta level towards the object level (arrow number (1) in Figure 1). Each change in the meta-active constraint store concerns the passive constraints of the object system. Therefore, the object level passive constraint store has to be modified (extended or simply changed) each time a meta-level active constraint changes. This linking rule is formalized by the following linking rule which acts on the active constraints of the meta level and produces new object level passive constraints:

$$\frac{C_m}{S_{obj} \cup S_{obj-m}}$$

where the constraint store S_{obj-m} is derived from the translation of the meta constraints in object level constraints.

The other linking rule (arrow number (2) in Figure 1) concerns the propagation in the meta system when passive constraints in the object system are modified. First of all, a transition must occur when a new passive constraint in the object system is encountered. This relation generates a meta active constraint which links

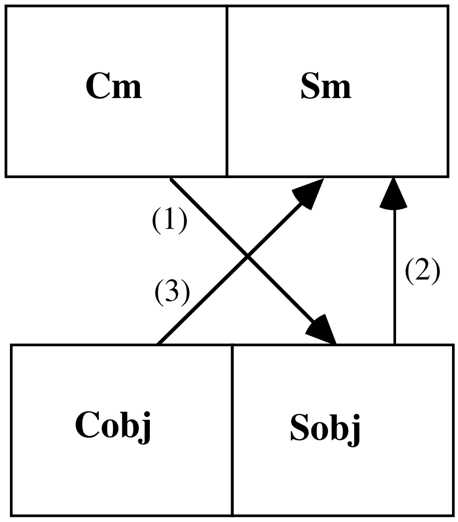

Figure 1: Transitions between the object and the meta constraint stores

a meta-variable to represent in its domain the relations that hold between the two object-variables occurring in the passive object constraint.

It is worth noting that as in the transition $\rightarrow_c$ in the two systems, when a new constraint is encountered, it should be first added to the passive constraint store. Then the transition $\rightarrow_i$ moves the active constraints from the passive to the active set and performs the propagation. Therefore, the transition $\rightarrow_c$ takes a constraint from the set of atoms and constraints A and adds it to the passive constraint store. In the same way, transition (2) in Figure 1 generates a meta-constraint starting from an object level passive constraint and adds this new active meta constraint to the passive meta constraint store. Then the transition $\rightarrow_i$ in the meta system will move this constraint to the active constraint store and perform the propagation.

The transition can be written as:

$$\frac{S_{obj}}{S_m \cup S_{m-obj}}$$

where the constraint store S_{m-obj} is obtained by the reification of the object level constraints into meta constraints.

The interactions between the object and the meta levels also involve the propagation of active meta-constraints when object active constraints do not entail some constraint any more (arrow number (3) in Figure 1). Therefore, the linking rule for the object to the meta system should add to the passive meta constraint store the constraints entailed by the values of the object level variables. Again, the transition $\rightarrow_i$ in the meta system assures that this constraint is moved into the active constraint store and performs the propagation. The corresponding linking rule has the form:

$$\frac{C_{obj}}{S_m \cup S'_{m-obj}}$$

where S'_{m-obj} is the constraint store entailed by the active object constraints[6].

This rule is a special purpose transition for the kinds of constraints we are considering.

This transition can be also described in terms of $\rightarrow_i$ transition of the object level system. In fact, each time an active constraint changes, we can directly add those binary relations entailed by the active constraints to the passive object constraint store. Then the linking rule which links S_{obj} to S_m (arrow number (2) in Figure 1) performs the reification of the new passive object constraints. However, the addition of a new transition to the object level system changes its behavior in some cases.

It is worth noting that each time a (meta or object level) constraint store changes due to the propagation of the other level constraints the system performs its own propagation in order to check the consistency and to infer new constraints.

3.3 Top-Down Execution

The operational semantics of our system is defined, according to [19], in terms of a transition system on *states*. In the previous Sections we have defined the primitive linking rules between each couple of levels of the system.

Now we should define our system behavior by providing one or more sequences of operators (transitions) to be applied to the states of the system which lead it to either a solution or a failure.

A derivation is a sequence of transitions:

$$\langle A_1, C_1, S_1\rangle \rightarrow \langle A_2, C_2, S_2\rangle \ldots \rightarrow \langle A_i, C_i, S_i\rangle \rightarrow \ldots$$

The initial state depends on the program to be executed. This meta architecture should be transparent to the user which writes a program as if the architecture contains only one level. Usually, the syntax of the programs is the same of the first level while queries are much more powerful. For example, if the first level is a finite domain solver the user writes common CLP(FD) programs. If we add to the solver further levels which are able to perform more powerful computations the user can

[6] For the sake of completeness, the linking rules (2) and (3) could be unified in a unique linking rule of the form

$$\frac{C_{obj} \cup S_{obj}}{S_m \cup S''_{m-obj}}$$

where $S''_{m-obj} = S_{m-obj} \cup S'_{m-obj}$.

query the system in order to get this information. Therefore, the initial state of the first level is $\langle A, \emptyset, \emptyset \rangle$, where A is the initial multiset of atoms and constraints, and the initial state of each meta constraint solver is $\langle \emptyset, \emptyset, \emptyset \rangle$.

However, there is the possibility of writing special purpose programs for other levels and their initial state in this case is $\langle A_m, \emptyset, \emptyset \rangle$, where A_m is the initial multiset of meta-atoms and constraints.

A state which cannot be rewritten further is called *final state*. The *final state* succeeds if it has the form $\langle \emptyset, C, S \rangle$. A derivation fails if it is finite and the final state is *fail*. A derivation flounders if it is finite and the final state has the form $\langle A, C, S \rangle$ where $A \neq \emptyset$.

The behavior of the two systems, i.e., the object and the meta level systems, can be described in terms of their derivations. In particular, let $\rightarrow_{ris} = \rightarrow_r \rightarrow_i \rightarrow_s$ and let $\rightarrow_{cis} = \rightarrow_c \rightarrow_i \rightarrow_s$. If the two systems can be described in terms of $\rightarrow_{ris}$ and $\rightarrow_{cis}$, they are called *quick checking*, according with the definition of [19].

If a transition starting from a state with a nonempty collection of atoms and constraints A either fails or contains a $\rightarrow_c$ transition or contains a $\rightarrow_r$ transition, the two systems are said to be *progressive*, according with the definition of [19].

It is worth noting that the two systems are not *ideal*. If a CLP system is *quick checking*, *progressive*, *infer* is defined as $infer(C, S) = (C \cup S, \emptyset)$ and the predicate $consistent(C)$ holds iff $\mathcal{D} \models \tilde{\exists}$C, then it is called *ideal*. Most constraint systems are *quick checking*, *progressive* but not *ideal*.

An interesting point concerns the transitions from the meta to the object level and viceversa. We should define a partial order between these transitions. Referring to the Figure 1 we should define transitions corresponding to the linking rules (1), (2) and (3). Let us start with the linking rule number (1).

When the meta level system performs an $infer$ and a *consistent* transition, i.e., $\rightarrow_{is}$, it produces new active consistent unary constraints. These constraints should be reflected at the object level via a transition, called $\rightarrow_{c'}$, and added to the object passive constraint store. This transition is very similar to $\rightarrow_c$ of the object system which takes a new constraint c from the set A and adds c to the passive constraint store. The only difference concerns the "source" of the constraint c which, in this case, comes from another level. After a $\rightarrow_{c'}$ transition the object level performs a $\rightarrow_{is}$ transition for processing the new constraints. Therefore, the $\rightarrow_{c'}$ transition can be written as:

$$\frac{\langle A_m, C_m, S_m \rangle \rightarrow_{is} \langle A_m, C'_m, S'_m \rangle}{\langle A_{obj}, C_{obj}, S_{obj} \rangle \rightarrow_{c'} \langle A_{obj}, C_{obj}, S_{obj} \cup S_{obj-m} \rangle}$$

where S_{obj-m} has been defined in Section 3.2.

The linking rules (2) and (3), which are conceptually separated, can be unified in a unique transition $\rightarrow_{c''}$ which translates active and passive object level constraint stores into passive meta level constraints. Therefore, after an $infer$ and *consistent*

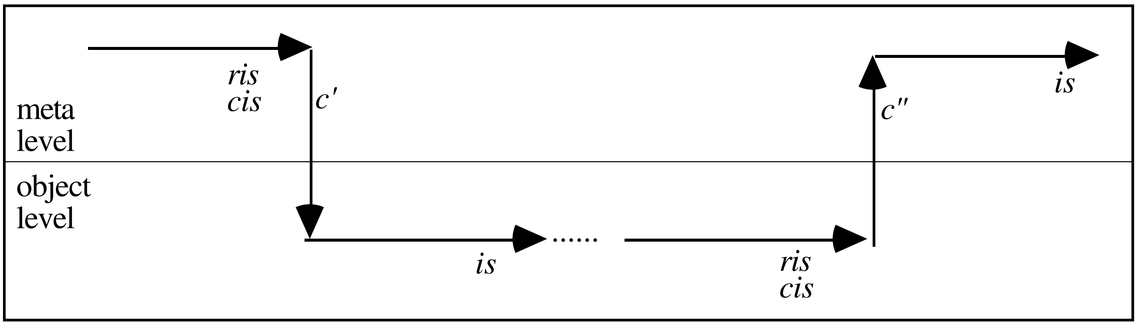

Figure 2: Transitions between the object and the meta systems

transition, i.e., $\rightarrow_{is}$ in the object level, the $\rightarrow_{c''}$ transition translates active and passive constraints stores in the meta passive constraint store.

$$\frac{\langle A_{obj}, C_{obj}, S_{obj}\rangle \rightarrow_{is} \langle A_{obj}, C'_{obj}, S'_{obj}\rangle}{\langle A_m, C_m, S_m\rangle \rightarrow_{c''} \langle A_m, C_m, S_m \cup S_{m-obj} \cup S'_{m-obj}\rangle}$$

where S_{m-obj} and S'_{m-obj} have been defined in Section 3.2 and represent respectively the reflection (reification) of S'_{obj} and C'_{obj}.

As for $\rightarrow_{c'}$, the $\rightarrow_{c''}$ transition can be considered a sort of $\rightarrow_c$ transition with the only difference that the constraints added to the passive constraint store are not taken from the set A but come from another level.

The partial order between the two level transitions is depicted in Figure 2. We can notice that also the interactions between the two systems can be described in terms of $\rightarrow_{c'is}$ and $\rightarrow_{c''is}$ transitions and so the overall system is also *quick checking*.

4 Consistency Algorithms

In this Section, we present a specialization of our meta CLP general scheme (for details see [24]). This specialization concerns a multi-level architecture for handling an explicit representation of constraints. In fact, meta levels maintain an explicit representation of constraints of the underlying object module and are mainly CLP(FD) solvers. By working on finite domains we are able to represent constraints explicitly by maintaining couples and in general n-tuples of values allowed by the constraints. This application of the meta-architecture allows us to exploit consistency algorithms. Consistency algorithms have been widely investigated by [16, 17, 22, 26, 28, 30].

This specialization allows to reach a certain level of consistency by using weaker algorithms, as suggested in [14]. Usually, a CLP(FD) solver propagates constraints with an arc consistency algorithm but several applications needs higher levels of consistency, such as, for example, temporal reasoning which needs a path consistency propagation algorithm.

This claim also motivated another work [6] where the path consistency algorithm has been embedded in a CLP(FD) constraint solver. We present a more general approach to the problem which allows us to reach any degree of consistency[7].

4.1 The Domains

The definition of the domains of the different levels of the architecture is straightforward in this case. The domain of the first level, according to [19], which is a CLP(FD) solver, is the following: let D=Z and

$$\Sigma = \{\{\in [m,n]\}_{m\leq n}, +, =, \neq, \leq\}.$$

The meaning of symbols is the same reported in Section 5.1.

If all the underlying systems are CLP(FD) solvers performing arc consistency as the propagation method, the level n has the following domain: D=Z^n where Z^n is the cartesian product of Z.

Otherwise, the domain of the level n depends on the degrees of consistency performed by the underlying systems. In [24] we have demonstrated that the level of consistency achieved by the whole system is the sum of the levels of consistency achieved by each level minus one.

If we have n levels each one obtaining a degree of consistency c_i with $i = 1..n$ the overall consistency achieved by the system is a k-consistency, where k is computed as

$$k = (\sum_{i=1}^{n} c_i) - 1$$

Therefore, $n+1$-th level has a domain D=Z^k. In fact, one level performs consistency on the already consistent k-tuples of values of the underlying system.

Example 4.1 *Suppose we have two CLP(FD) performing an arc consistency algorithm. This means that for each couple of variables the algorithm, i.e., the predicate* infer, *deletes from their domains values which are not consistent with at least one value of the other variable domain. Therefore, suppose we have variables X, Y and Z ranging on the domain $\{1..2\}$. The constraints between them are: $X \neq Y$, $X \neq Z$ and $Y \neq Z$. Intuitively, the constraints are inconsistent with the values of the variables, but an arc consistency algorithm does not remove any value from the domains.*

[7] It is worth mentioning that several applications (in the field of temporal reasoning) need a 4-consistency algorithm. Moreover, the work presented in [15] finds sufficient conditions for backtrack free search, i.e., sufficient consistency algorithms for finding a solution without searching.

By reifying the constraint in the meta-level, we obtain three meta-variables XY, XZ, and YZ ranging on the domain [(1,2),(2,1)] which represent the arc-consistent couples allowed by constraints of the object system. The three meta-variables are linked by a constraint $XY +_m YZ =_m XZ$ where $+_m$ and $=_m$ are the addition and equality symbol in the meta system. The propagation of the above-mentioned meta-constraint produces the domain for XZ: [(1,1),(2,2)] [8] *which should be intersected with the previous domain of XZ producing an empty domain. Therefore, the constraints are not path consistent.*

Again, suppose we have two levels performing a path-consistency algorithm. The overall consistency reached is a k-consistency with $k = 5$. The second level reasons on the consistent triples given by the path consistency algorithm of the first level. The second level performs a path consistency on these triples and, therefore, finds all the consistent paths linking three triples (three 3-length paths, i.e., 5 variables). The third level, therefore, acts on 5-tuples and so on.

The signature of the n-th meta-level is again

$$\Sigma_n = \{\{\in C\}, +_n, =_n, \neq_n, \leq_n\}.$$

where symbols in Σ_n are interpreted by the Σ_n-structure $\mathcal{D}_n$ in the following way: $\in C$ is interpreted as a unary constraint symbol where C is a subset of D=Z^k, i.e., is a set of k-tuples allowed by constraints of the underlying system. The symbol $+_n$ is the sum (or composition) symbol of k-tuples of the n-th level. The symbol $=_n$ and $\neq_n$ are interpreted in the usual way, while $\leq_n$ depends on the specific application.

One of the main advantages of the framework is its flexibility. We can reach any degree of consistency by changing the numbers of levels, using lower degree algorithms.

In this way, we can use weaker and simpler algorithms in order to reach a higher level of consistency. For example, a CLP(FD) solver performs an arc consistency algorithm for the propagation of constraints. If we need a higher level of consistency, we do not have to modify the solver, it is sufficient to add as many levels as we need to reach the consistency desired.

4.2 The Operational Semantics

The operational semantics of the system is defined in terms of "top down execution" of the two components and the linking rules between the two systems.

First of all, we define what are the active and passive constraint stores of the different levels. The active constraint store contains in each level the unary constraints which link variables to take their values from a finite domain of k-tuples. Therefore, each variable can be instantiated to a k-tuple when the constraint is solved.

[8] The meta addition of (1,2) $+_m$ (2,1) produces the couple (1,1).

The passive constraints link the meta-variables to be constrained with binary meta-constraints and operations defined in the signature.

The very important peculiarity of the operational semantics concerns the predicate *infer* which can perform whatever kind of consistency algorithm.

An interesting point concerns the linking rules between two systems. For the sake of clarity in the examples we will refer to two levels each performing an arc consistency algorithm.

The first linking rule (arrow number (1) in Figure 1) links the meta active constraint store to the object level passive constraint store. It has the form:

$$\frac{C_m}{S_{obj} \cup S_{obj-m}}$$

where S_{obj-m} represents the passive object level constraint store entailed by the k-tuples of the meta level. Suppose we have a two level architecture where each level performs an arc consistency: the meta level reasons on couples of values of the underlying system. Suppose now we have a meta-variable XY ranging on a domain $D_{XY} = \{(3,4),(4,6),(2,8)\}$ (meta-active constraint) and the variables X and Y are linked by the passive object level constraint $X \leq Y$. The constraint equal is no more entailed by the couples of consistent values belonging to D_{XY}.

Moreover, this linking rule has a more powerful meaning: in fact, it can add unary object level constraints containing values entailed by the meta domains to the passive object level constraints. Suppose that a meta constraint propagation leads the meta-variable XY to range on the domain $D_{XY} = \{(3,4),(4,6),(2,8)\}$ and the domain of X contains values $\{2,3,4,7\}$. The value 7 should be deleted by the domain of X.

The second linking rule (arrow number (2) in Figure 1) concerns the propagation of meta active constraints when the object level passive constraints are modified. The new constraints produced by this propagation are added to the passive constraint store even if they are active. They will be transformed in active constraints by the propagation mechanism of the meta level. This linking rule corresponds to:

$$\frac{S_{obj}}{S_m \cup S_{m-obj}}$$

where S_{m-obj} represents the unary meta constraints containing the k-tuples consistent for the underlying system. Therefore,

$$S_{m-obj} = \bigcup_{\forall x_i \in D_i, x_j \in D_j} x_{ij} \in D_{ij}$$

where $D_{ij} \subseteq D_i \times D_j$. A binary constraint, in fact, defines a subset of the cartesian product of the domains of two variables.

Suppose we have two object level variables X ranging on $D_X = \{3..5\}$ and Y ranging on $D_Y = \{4..6\}$. The linking rule ensures that the store S_{m-obj} contains the constraint $XY \in D_{XY}$ where

$$D_{XY} = \{(3,4),(3,5),(3,6),(4,5),(4,6),(5,6)\}.$$

The third linking rule (arrow number (3) in figure 1) is represented by:

$$\frac{C_{obj}}{S_m \cup S'_{m-obj}}$$

where S'_{m-obj} is the constraint store entailed by the active object constraints. The meta-variables domain should always contain values consistent with the underlying level. Therefore, if the propagation of constraints leads to the canceling of a value from an object variable domain, all the k-tuples containing this value in the meta active constraint store should be deleted as well.

For example, if the domains of X and Y contain respectively the values $\{3..4\}$ and $\{4..5\}$ and the constraint between them is $X < Y$, the meta variable associated is XY ranging on a subset of the cartesian product of their domains, i.e., $\{(3,4),(3,5),(4,5)\}$. If the value 3 is deleted from the domain of X, the propagation of the linking rules should delete the couples $(3,4)$ and $(3,5)$. It is worth noting that this propagation leads another linking rule (number (1)) to delete the value 4 from the domain of Y.

5 Qualitative Reasoning

In this Section, we show the real applicability of the general meta CLP scheme by presenting a possible specialization concerning qualitative reasoning [23]. The pure CLP scheme does not suffice for this application. In fact, even if CLP has very efficient constraint propagation and handling mechanisms, it is not able to reason on constraints and, therefore, it is not able to perform qualitative reasoning.

We specialize the general scheme in order to build a two-level architecture where the first level is a CLP(FD) and the second is a meta constraint solver which reasons on constraints of the first. The second level reasons on finite domains of sets of constraints, performs operations on them such as composition, intersection and propagates meta constraints such as tighter relations[9].

This specialization is necessary when one wants to know if a particular set of qualitative constraints is consistent or find the minimal network as happens, for example, in temporal reasoning applications.

[9] A constraint C' is tighter than C" if every pair of values allowed by C' is also allowed by C".

5.1 The Domains

As already said, the object level constraint solver is a CLP(FD) component. Therefore, its domain, according to [19], is the following: let D=Z and

$$\Sigma = \{\{\in [m,n]\}_{m \leq n}, +, =, \neq, \leq\}.$$

The symbols in Σ have their usual meaning apart from $\{\in [m,n]\}_{m \leq n}$ which is defined as follows: for every pair of integers m and n, $\mathcal{D}$ interprets the interval constraint $x \in [m,n]$ as $m \leq x \leq n$. $\mathcal{L}$ is the class of constraints which is generated by the primitive constraints where each variable is subject to an interval constraint. $\mathcal{FD} = (\mathcal{D}, \mathcal{L})$ is the constraint domain. A typical constraint which can be expressed in this domain is

$$x \in [3,7] \wedge y \in [1,9] \wedge x \neq 5 \wedge 4x - y \leq 16$$

As concerns the meta-level, the domain should contain the constraints of the underlying system. Therefore, the qualitative constraint solver is based on a meta-constraint solver which uses finite domains of relations. A meta-variable R_{XY} is automatically generated each time a new qualitative constraint between object level variables X and Y is introduced. Qualitative constraints are disjunctions of primitive constraints between variables. Therefore, if between the variables X and Y we have a relation of the kind:

$$(X\ r_1\ Y) \vee \ldots \vee\ (X\ r_n\ Y)$$

which can be written as:

$$X\{r_1, \ldots, r_n\}Y$$

where r_is are primitive qualitative relations[10], then the system automatically creates a meta-variable R_{XY} whose domain D_{XY} is the powerset of $\{r_1, \ldots, r_n\}$.

More formally, let D be the set of possible constraints such as, for example, constraints between integers: D=$\{<, =, >\}$. Let $\mathcal{P}$(D) be the powerset of D, i.e., the set containing all the possible subsets of D[11]:

$$\mathcal{P}(\mathrm{D}) = \{\{\}, \{<\}, \{>\}, \{=\}, \{<,=\}, \{<,>\}, \{>,=\}, \{<,=,>\}\}.$$

Now let V be a subset of $\mathcal{P}$(D), i.e., V$\subseteq \mathcal{P}$(D), the signature of the meta-level system is

$$\Sigma_m = \{\{\in V\}, +_m, =_m, <_m\}^{12}.$$

[10] With primitive relations we mean primitive constraints which are elements of $\mathcal{L}$ in the finite domain framework.

[11] It is worth noting that the powerset of any set of constraints always contains the empty set $\{\}$. For the sake of readability we omit it throughout the paper.

[12] The subscript m in symbols of Σ has been introduced in order to distinguish these operations in the object system and in the meta system.

The symbols in Σ_m are interpreted by the Σ-structure $\mathcal{D}_m$ in the following way. The symbol $\{\in V\}$ is defined as follows: for each V$\subseteq \mathcal{P}$(D), $\mathcal{D}_m$ interprets the constraint $r \in V$ as the usual instance of the set V, therefore, r can assume one value among all the possible elements of V (subsets of constraints). The symbol $+_m$ is interpreted by $\mathcal{D}_m$ as the composition operator. The composition of two meta-variables R_{XZ} and R_{ZY} admits only those values r_{xy} for which there exists $r_{xz} \in D_{XZ}$ and $r_{zy} \in D_{ZY}$ such that $r_{xy} = r_{xz} \otimes r_{zy}$ where the symbol $\otimes$ represents the composition of primitive constraints and is defined by a transitivity table on the constraints of D.

The symbol $=_m$ is interpreted by $\mathcal{D}_m$ as the intersection of constraints and, in particular, it unifies two variables in the domain of $\mathcal{P}$(D). The intersection between two constraints $X\{r_1, \ldots, r_n\}Y$ and $X\{r'_1, \ldots, r'_n\}Y$ is the powerset of the set theoretic intersection between the two vectors $\{r_1, \ldots, r_n\}$ and $\{r'_1, \ldots, r'_n\}$. A meta-variable representing the possible relations between X and Y is associated with each constraint. Therefore, the intersection operation can be performed by imposing the two meta-variables R_{XY} and R'_{XY} (corresponding to the two constraints) to be equal. In normal constraint solvers, imposing the equality of two variables requires a meta-unification that can be handled and managed by the constraint solver by, for example, intersecting the variable domains.

The symbol $<_m$ is interpreted as the relation "tighter" among constraints. A set of constraints C' is tighter than another set of constraints C" if each pair of values allowed by C' is also allowed by C". In our case, the relation tighter can be translated in the set inclusion. This relation is a partial order relation on the elements of the domain. In fact it defines a lattice where the bottom is $\perp = \{\}$ and the top is $\top = \mathcal{P}$(D). Therefore, a constraint imposing a relation on non-comparable elements should fail.

5.2 The Operational Semantics

In this Section, we specialize the operational semantics of the two components and the interactions between them.

As regards the operational semantics of the two components, we have to define the operations *infer* and *consistent* for the two solvers.

In the finite domain solver (the object level), the *infer* operation performs an arc consistency algorithm. The arc consistency is a weak constraint propagation algorithm which ensures that every arc of the network is consistent. An arc, linking variables X and Y, is consistent if for each value in the domain of X there exists at least one value in the domain of Y which is consistent with the constraint linking the two variables. The predicate *consistent* should be applied to the active constraint store and holds if no variable domain is empty.

In the meta constraint solver, the two predicates have the same structure: the predicate *infer* performs the arc consistency on the meta-variables by eliminating the constraints symbols which cannot satisfy meta-constraints and operations. The predicate *consistent* holds if no meta-variable domain is empty.

The most interesting part of the operational semantics concerns the interactions between the two systems. The linking rules between the two systems are defined in Figure 1 Arrow number (1) refers to the linking rule:

$$\frac{C_m}{S_{obj} \cup S_{obj-m}}.$$

For each unary constraint $r_{xz} \in D_{xz}$, $D_{xz} \subseteq \mathcal{P}(D)$ in C_m

$$S_{obj-m} = \bigcup_{r_{xz} \in D_{xz}} lub(r_{xz}).$$

For instance, suppose that after the computation of $infer(C_m, S_m)$ the set of active meta constraints C'_m contains the constraint:

$$r_{xz} \in [\{<\}, \{=\}, \{<, =\}, \{\}].$$

The lub of the variable's domain is $\{<, =\}$, and this lub represents the constraints that hold between the object variables x and z in the object system.

For the sake of completeness, if a meta variable r_{xz} has an associated domain containing only the empty set, i.e., $r_{xz} \in [\{\}]$, it means that a failure in the object system occurred. This object level failure can be either propagated in the meta system (as a failure) or handled by the meta-reasoner in order, for example, to retract some constraints. On the other hand, a failure in the meta-system corresponds to an empty domain of a meta-variable, i.e., $r_{xz} \in [\ ]$.

Arrow number (2) in figure 1 concerns the linking rule:

$$\frac{S_{obj}}{S_m \cup S_{m-obj}}$$

where

$$S_{m-obj} = \bigcup_{\forall x,y \in D,\ x\{r_1,r_2,\ldots,r_n\}y,\ r_1,r_2,\ldots,r_n \in \mathcal{L}} \mathcal{P}(r_1, r_2, \ldots, r_n)$$

For instance suppose that the set of passive object constraints S_{obj} contains the constraint $x \leq y$ which can be written as $x\{<, =\}y$, the transition creates a meta-variable r_{xy} ranging on the powerset of $\{<, =\}$.

The third linking rule (arrow number (3) in Figure 1) adds to the passive meta constraint store those constraints entailed by the values of the object level variables and has the form:

$$\frac{C_{obj}}{S_m \cup S'_{m-obj}}$$

where S'_{m-obj} is the constraint store entailed by the active object constraints. This rule is a special purpose transition for the kind of constraints we are considering. For example, if the domains of X and Y contain respectively the values [1..10] and [14..24] and the constraint between them is $X \leq Y$ the constraint $=$ is no longer entailed by the values in their domain. Therefore, the constraint should be propagated which eliminates $=$ from the possible values for the meta variable R_{XY} (possible constraints between object variables X and Y).

6 Conclusions and Future Works

In this paper, we have presented a general meta CLP scheme which can be easily built by adding constraint solvers each reasoning on the constraints of the underlying level. This architecture allows us to increase the expressive power of CLP and inherits all the advantages of a multi-level meta architecture such as modularity, expandibility and flexibility.

CLP itself is defined as a general scheme which can be specialized on a particular domain. A CLP language, in fact, can be defined by specifying the domain and the operational semantics. In the same way, we define a general meta CLP scheme which can be specialized on different domains by defining the domain and the operational semantics of each level composing the architecture. In particular, the operational semantics should be given in terms of the operational semantics of each level and in terms of the linking rules between two adjoining levels.

Finally, the top down execution defines the way in which the transition rules should be applied during the computation.

We have presented, together with the general scheme, two specializations of the scheme itself. The first concerns a multi-level architecture capable of dealing with any degree of consistency not by increasing the difficulty and the power of the algorithm, but just by adding further levels performing algorithms of lower degrees of consistency.

The second specialization concerns the possibility of performing qualitative reasoning with CLP by adding a meta level reasoning on the constraints of the underlying level. This specialization has been applied to the field of temporal reasoning. In fact, in [23] we have shown that a meta CLP architecture can be easily used for integrating qualitative and quantitative temporal reasoning.

We have developed a prototype version of the two specializations by using an extension of ECL^{i}PS^e [13] finite domain library (see [23, 24] for details). Our research is now concentrated on finding new interesting specializations on real numbers, boolean, sets and Herbrand Universe. We are also investigating another application of the system to qualitative reasoning by using a CLP(Intervals) [3] as an object-level and the Allen Interval Algebra [2] as the meta-level.

Moreover, we will embed in the meta constraint solver the capability of making some recovery actions on the lower level system. In fact, the meta level is able to detect some inconsistencies of the low level system and can therefore maintain some previous consistent situations in order to recovery from a failure. We will investigate the possibility of using some Assumption Truth Maintenance System techniques [8] for handling this kind of problems.

References

[1] L.Aiello, C.Cecchi, D.Sartini, "Representation and Use of Metaknowledge" in *Proceedings of the IEEE*, vol. 74, No. 10, 1986, pp. 1304–1321.

[2] J.F.Allen, "Maintaining Knowledge About Temporal Intervals", in *Communications of the ACM*, vol. 26, 1983, pp. 832–843.

[3] F.Benhamon, D.McAllester, P.Van Hentenryck, "CLP(Intervals) Revisited", *Tech. Report* CS-94-18 Computer Science Department, Brown University, 1994.

[4] K.A.Bowen, R.A. Kowalski, "Amalgamating Language and Metalanguage in Logic Programming" in *Logic Programming*, K.Clark and S.Tarnlund Eds., Accademic Press NY, 1982, pp. 153–173.

[5] CHIP User's Guide. Version 4.0. A. COSYTEC SA 1993 France.

[6] P.Codognet, G.Nardiello, "Path Consistency in clp(FD)", in *Proceedings of the First International Conference Constraint in Computational Logics CCL94*, 1994, pp. 201–216.

[7] A. Colmerauer, "An Introduction to Prolog III" ACM Communication July 1990, 33(7), pp70–90.

[8] J. de Kleer, "An Assumption-based TMS", in *Artificial Intelligence*, n.28, vol 3, 1986, pp. 127–162.

[9] M.Dincbas, P.Van Hentenryck, M.Simonis, "Solving Large Combinatorial Problems in Logic Programming", in *EURO-TIMS Joint International Conference on Operational Research and Management Science*, France, July, 1988.

[10] M.Dincbas, P.Van Hentenryck, M.Simonis, "Solving the Car Sequencing Problems in Constraint Logic Programming", in *ECAI88 European Conference on Artificial Intelligence*, Germany, August, 1988.

[11] M.Dincbas, P.Van Hentenryck, M.Simonis, A.Aggoun, T.Graf, F.Berthier, "The Constraint Logic Programming Language CHIP" , in *Proceedings of the International Conference on Fifth Generation Computer System (FGCS88)*,1988, pp.693–702.

[12] M.Dincbas, P.Van Hentenryck, M.Simonis, "Solving Large Combinatorial Problems in Logic Programming", in *Journal of Logic Programming*, 8 (1-2) pp.75–93.

[13] ECL^{i}PS^e User Manual Release 3.3, ECRC 1992.

[14] E.C. Freuder, "Synthesizing Constraint Expressions", in *Communication of the ACM*, vol. 21, 11, 1978, pp. 958–966.

[15] E.C. Freuder, "A Sufficient Condition for Backtrack Free Search", in *Communication of the ACM*, vol. 29, No1, 1982, pp. 24–32.

[16] E.C. Freuder, A.K. Mackworth, "The Complexity of some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems", in *Artificial Intelligence*, vol.25, 1885, pp.65–74.

[17] E.C. Freuder, A.K. Mackworth, "Special Issue on Constraint Based Reasoning", *Artificial Intelligence*, vol. 58, 1992.

[18] J.Jaffar , J.L.Lassez, "Constraint Logic Programming", in *Proceedings of the Conference on Principle of Programming Languages*, Munich 1987.

[19] J.Jaffar, M.J.Maher, "Constraint Logic Programming: a Survey", in *Journal of Logic Programming on 10 years of Logic Programming*, 1994.

[20] J.Jaffar, S.Michaylov, P.J.Stuckey, R.H.C.Yap, "The CLP(R) Language and System", ACM Transaction on Programming Language and Systems, vol 14(3), pp.339–395, 1992.

[21] R.Kowalski, "Logic for Problem Solving", North-Holland,1979.

[22] V. Kumar, "Algorithms for Constraint-Satisfaction Problems: A Survey", in *AI Magazine*, vol. 13, 1992, pp. 32–44.

[23] E.Lamma, P.Mello, M.Milano, "A Meta Constraint Logic Programming Architecture for Qualitative and Quantitative Temporal Reasoning", Technical Report DEIS-LIA-95-001, 1995.

[24] E.Lamma, P.Mello, M.Milano, "Exploiting Consistency Techniques Via a Meta Constraint Logic Programming Architecture", Technical Report DEIS-LIA-95-004, 1995.

[25] J.W.Lloyd, "Foundation of Logic Programming", Second Extended Edition, Springer-Verlag 1987.

[26] A.K. Mackworth, "Consistency in Networks of Relations", in *Artificial Intelligence*, vol.8, 1977, pp. 99–118.

[27] P.Maes, "Computational Reflection", Tech. Report AI-lab VUB, Brussels, 1987.

[28] R.Mohr, T.C.Henderson, "Arc and Path Consistency Revisited", in *Artificial Intelligence*, vol.28, 1986, pp. 225–233.

[29] A.Montanari, "Networks of Constraints: Fundamental Properties and Applications to Picture Processing", in *Information Sciences*, vol.7, 1974, pp. 95–132.

[30] B.Nadel, "Tree Search and Arc Consistency in Constraint-Satisfaction Algorithms", in *Search in Artificial Intelligence*, eds. L.Kanal and V.Kumar, Springer Verlsg, NY, 1990, pp.287–342.

[31] J.F.Puget, "A C++ Implementation of CLP", Technical Report 94–01, ILOG Headquarters, France.

[32] P.Van Hentenryck, "Constraint Satisfaction in Logic Programming", MIT Press, 1989.

[33] P.Van Hentenryck, Y.Deville "Operational Semantics of Constraint Logic Programming over Finite Domains", *Tech. Report* CS-90-23 Computer Science Department, Brown University, 1990.