CHOCO: implementing a CP kernel

François Laburhe and the OCRE project team

Bouygues e-Lab
1 avenue Eugène Freyssinet
78061 St. Quentin en Yvelines, France
flaburth@challenger.bouygues.fr; ocre@emn.fr

This paper presents CHOCO, a Constraint Programming (CP) library, developed within the OCRE project, as the kernel of a platform for CP research in combinatorial optimization. CHOCO features the standard utilities of a CP System, such as propagation events, data structures implementing domains, filtering algorithms or support for tree search. CHOCO has been implemented with two specific additional requirements in mind. First, the software architecture is modular in order to easily support extensions; second, the management of propagation events has been thoroughly optimised.
The paper reviews some of CHOCO's main design decisions, presents the software architecture and discusses the policy for managing and scheduling propagation events: we show how the standard policy from arc consistency can be specialized in the cases of arithmetic and global constraints.

1. Introduction

CHOCO is a library implementing basic primitives for constraint programming : domain management, constraint propagation, as well as global and local search procedures. CHOCO serves as the kernel of a broader project, OCRE[1] intended to building a common software platform for CP among several research teams (in French, O.C.R.E. is an acronym for *Constraint Tool for Research and Education*). It is a joint project between Ecole des Mines de Nantes, ONERA Centre de Toulouse, Université de Montpellier LIRMM, INRA and Bouygues e-lab. It started from the fact that each participant often conducted experiments with CP algorithms and models, but took little advantage of the work conducted in the other teams, for lack of a common software platform. OCRE provides every participant with the ability to easily compare, combine and experiment with algorithms and methods proposed by other researchers. This academic platform targets research and education; CHOCO is its ground layer, and is available as free software[2]. Such a ground position induces a few requirements on CHOCO, which has to be:

– *Small and Simple*, so that everyone working in OCRE can understand and extend it, and so that it may be used for teaching purposes.
– *Efficient*, so that it will not penalize OCRE algorithms by any run-time overhead.

[1] www.emn.fr/dept_info/recherche/equipes/contraintes/ocre/public/Welcome.html
[2] covered by the General Public Licence from the Free Software Foundation

– *Generic*, so that further OCRE components, for instance, extending the scope to, say, other constraint domains, can be compatible with the CHOCO kernel. The right balance between being generic and being efficient has to be found.

There is actually no new functionality in CHOCO, compared to the usual CP tools (no glorious global constraint, no new consistency algorithm, no new search paradigm) : CHOCO simply features the set of CP utilities that one has to go through when implementing a new constraint system from scratch. Indeed, an implementer starting such a CP system project for experimenting with an algorithm must not only code many utilities that have nothing to do with the targeted algorithm, he is also left with very little help from the literature, as implementation has rather been a disregarded topic in the CP research community. For these two reasons, we find it worthwhile to present the general design decisions that one is faced with while implementing a CP system. This paper should thus be read as a practical implementation report and we hope that it will be helpful for other researchers. A future paper will present comparisons and figures to evaluate some of the design decisions against implementation alternatives.

The paper is organized as follows. Section 2 presents a few basic design decisions that oriented the implementation. Section 3 quickly reviews the object oriented architecture and discusses the role of a few abstract classes. Section 4 is the main contribution: we present the policy for scheduling propagation events. It is an adaptation of standard rules from arc consistency algorithms, based on few practical considerations.

2. Design decisions

2.1 User interface

The first major design decision concerning the construction of a CP system is the choice between a library and an autonomous system. Ilog Solver belongs to the first class, while OPL Studio, Oz or CHIP belong to the latter. Building a library eases the task of application integration within computer environments (GUI, databases, etc.), while implementing a full fledged system generally yields more elegant problem statements, raising the abstraction level from programming languages to modelling languages.

CHOCO is a library, for the sake of ease of integration. Moreover, it is written in CLAIRE [Caseau, Laburthe 1996], a programming language[3] designed for expressing complex discrete algorithms in an elegant manner. As any CLAIRE library, CHOCO may either be used as is (as CLAIRE source code) or it can be compiled into a C++ library or a Java library. Therefore, CHOCO features some of the advantages of both solutions: CHOCO can be used as a CLAIRE source library with the CLAIRE interpreter for rapid and elegant prototyping of the application (the combination of CLAIRE and CHOCO approaches the level of a modelling language such as OPL). Afterwards, during deployment, the programmer can use the CLAIRE compiler to generate Java (or C++) code from her application and from the CHOCO library: the integration within an

[3] see http://www.ens.fr/~caseau/claire.html or http://www.clairelanguage.com

existing information system is almost as easy as with any Java (or C++) source library. Throughout this paper, all examples will be given in CLAIRE code.
The second design decision that we had to make was to decide what an application using CHOCO should look like. A small example with three variables and three constraints is displayed below.

```
let p := makeProblem("TRICS Toy Problem"),
    a := makeIntVar("A", 1, 100),
    b := makeIntVar("B", 2, 7),
    c := makeIntVar("C",-5,12) in
 ( post(p, a >= 3 * b - c),
   post(p, 2 * a - 6 * b == 5 * c),
   post(p, (a <= 10) implies (b + c > 14)),
   solve(p,true) )
```

Figure 1: a sample of CHOCO code

Three important decisions are illustrated on this small example:

- Encapsulate all information in a *Problem* object, rather than global structures. Doing so supports the definition of several *Problem*'s across a single session, which is useful for comparing models or solving a sequence of instances.
- Use operators (+,>,…) for defining arithmetic constraints. Since operators may not be redefined in Java; stating arithmetic constraint is less convenient in Java than in CLAIRE or C++. This is the main API difference between the CLAIRE and Java versions of CHOCO; the remainder of the API is identical.
- Clearly separate modelling (defining variable and stating constraints) from problem solving (propagation and search). Doing so has two advantages. On the one hand, it leaves the possibility to the user to edit a set of constraints, for instance, in order to apply simplification rules, before any propagation is performed. On the other hand, this allows the user to state all the constraints before performing heavy propagation, instead of performing it after each constraint addition.

2.2 Tree search

Concerning search trees, two important implementation decisions were made :

1. keep the computation in a non-distributed environment. For the sake of simplicity, the execution of a program yields only one process and does not use threads.
2. use a trailing stack for implementing backtrack [Aggoun, Beldiceanu 1991]. Each time a backtrackable data structure x (say the lower bound field of a domain variable) is updated from value a to value b, we store the reference to x and the former value a in a stack, before performing the assignment $x := b$. Upon backtracking, we pop the pairs from this stack and restore the original assignments $x := a$. This choice is motivated by memory usage (the alternative would be to copy the whole state at each node of the tree). Moreover, the trailing mechanism comes with the standard CLAIRE language: as soon as a field f is declared as "backtrackable", all updates to f are automatically performed with a copy of the old value on the trailing stack.

2.3 A simple object model

We now introduce the entity-relationship model on which CHOCO is based. Problems, domain variables, domains and constraints are modelled as objects: each type of variable (over integers, reals, trees, …) is implemented by a class inheriting from *AbstractVar*, each type of constraint ($x >= y$, $x <> y$, linear combinations, boolean combinations, constraints as n-tuples of feasible values, etc.) is implemented by a class inheriting from *AbstractConstraint*. The constraint network is modelled as a set of links between constraints and variables. Each constraint object contains its variables in some fields; each domain variable features a field with the list of constraints that it is involved in. Up to now, CHOCO is limited to integer valued variables; however, the architecture has been designed to easily support extensions.
The basic step of propagation consists in waking up all constraints linked to a variable, each time the domain of that variable is modified. Since the set of available constraint classes can be extended, we do not know *a priori* the type of constraints linked to a variable and the list of constraints attached to a variable is a heterogeneous list containing any descendents from *AbstractConstraint*. Therefore, a generic (virtual) method is called on all the constraints. Each constraint class provides an actual implementation of this propagation method.
The use of virtual methods (the notion of polymorphic functions in object languages) is at the very heart of CP systems. They offer a homogeneous framework for integrating constraints which supports a modular organization of the code [Puget 92]. Firstly, constraints can be implemented separately for two reasons: they can be propagated independently and in any order, by calling a virtual propagation method and secondly, constraints only need information on the state of their own variables (at least, in the case of local consistency algorithms).
The choice of such a software architecture with virtual methods and dynamic dispatching has two impacts on the source code:

- The notion of constraint can be defined as an abstract data type with an interface: a constraint is a data structure that implements a set of virtual methods for propagation.
- The constraint system can be described, from an operational point of view, in terms of events and agents: during propagation, each time a variable domain is shrunk (an event on that variable), the constraints involving that variable are waken (just as agents or as demons) and, in turn, they may generate new events (filtering out some values from domains). This framework is well adapted for solvers implementing forms of local consistency (arc consistency or weaker notions), and not for forms of consistency involving several constraints at a time. The only notions of strong consistency are implemented as global (n-ary) constraints.

2.4 A reactive system with events

The subtlety of CP algorithms and systems often resides in the efficient management of propagation events. The standard CSP formalism of constraints as tuples of values (without semantic) considers uniform events (value removals). Works in the CLP literature provide specific propagation algorithms from the constraint semantics: for instance, they derive updates on the bounds of the domains for arithmetic constraints.

In the CSP literature, the mechanism for waking up a constraint consists in computing an arc-consistent state after one or several value removals: there is thus only one filtering procedure and the global propagation loop is directed by constraints. In the CLP literature, the filtering algorithm is specialized along several propagation paths and the global propagation loop is directed by the domain reductions.
Thus, a general tool, supporting both spirits must manage various types of propagation events and offer the user the possibility to parameterize the control over propagation.
CHOCO manages four types of events and offers several modes for waking up constraints. By default, all events are used and the waking mode has been empirically selected for each type of constraints. However, the system can be controlled by the user who can decide to select simpler frameworks (basic AC-4, considering only bound events, removal of some optimisations, etc.)
CHOCO considers the following propagation events for finite domain variables:

- `INCINF`: the domain lower bound for some variable `v` is increased from `b` to `a`,
- `DECSUP`: the domain upper bound for some variable `v` is decreased from `b` to `a`,
- `INSTANTIATE`: the domain of some variable `v` is reduced to {`a`},,
- `REMOVAL`: the value `a` is removed from the domain of some variable `v`.

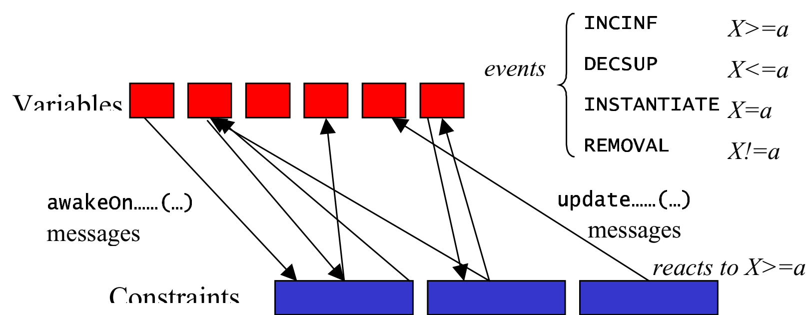


Figure 2: CHOCO ‘s architecture in terms of agents and events.

The figure above illustrates the message passing activity between variable and constraints: events are generated on variables, which inform their constraints (calling *awakeOn…()* functions). Constraints react to the events and in turn, generate new events (calling *updateOn…()* functions). and As displayed on the figure above, each time a propagation event occurs, the domain of the variable is updated appropriately and all constraints connected to that variable react to the event.

3. Software architecture

This section rapidly presents the software architecture of CHOCO with its class hierarchy and the generic services associated to each class. The current object hierarchy of CHOCO currently features around 30 classes. Inheritance is used to share services across two different implementations of a same notion: one way of

controlling propagation consists in creating subclasses of variables and constraints and redefining their own propagation management functions.
A few of them are abstract classes (with no instances) who implement abstract data types. They are worth mentioning because they amount to the low-level interface of a constraint programming system.

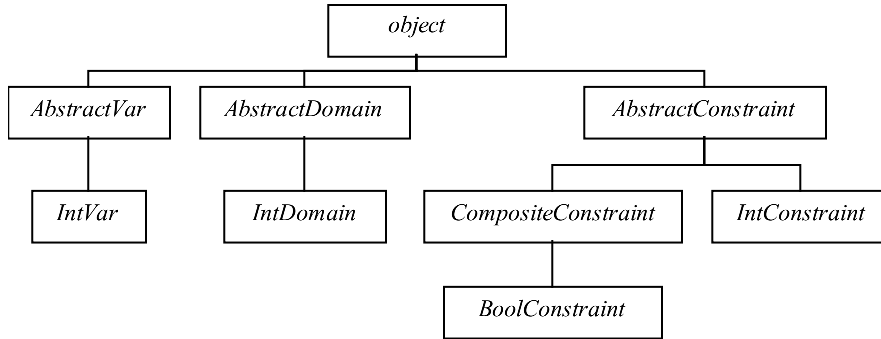


Figure 3: CHOCO's class hierarchy

Among these abstract classes:

- The *AbstractVar* class is the root class for all domain variables. It implements links to all constraints in which the variable is involved as well as information indicating how the variable occurs in the constraints. For instance, when *x.constraints[i]=c* and *x.indices[i]=3*, then *c* is the ith constraint in which *x* occurs and *x* is the third variable in *c* (see figure 4).

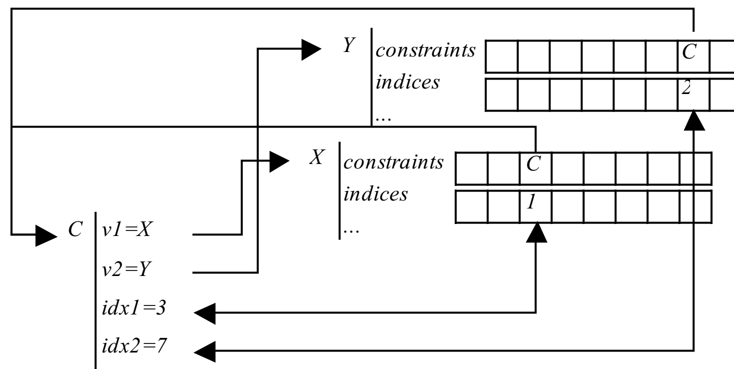


Figure 4: Encoding of the network in the fields of variable and constraint objects

- The *AbstractDomain* class is the root class for all domain data structures. It implements methods for removing values and for testing whether values are present or not in the domain.
- The *AbstractConstraint* class is the root class for all constraints. All classes inheriting from it implement :

– an indexed access to its variables. For instance binary constraints will implement two fields *v1* and *v2* with references to their two variables.
– a reference for finding back the constraint among all constraints linked to one of the variables. For instance, binary constraints implement two fields *idx1* and *idx2* such that *c.v1.constraints[c.idx1] = c*. The *idx1* field thus contains the index of *c* back among all constraints associated to its first variable (see figure 4).
– a feasibility check (to be called when all variables are instantiated).
– a satisfaction test, returning *true*, *false* or *unknown*, to test whether a constraint is surely satisfied, surely violated or may be anything, given the current status of the domains.
– a service for connecting and disconnecting a constraint from the network. Constraints are connected to the network right after being posted to the problem; they are disconnected as soon as they become satisfied.
– a service for transforming a constraint into its converse logical condition. This is used by Boolean combinations; global constraints which are too heavy to be combined by Boolean operators return an error when asked their converse constraint.

- The *CompositeConstraint* class is the root class for all constraints which are made up from several smaller constraints (such as Boolean combinations). It implements services for giving a unique global index to each occurrence of a variable in one of the subconstraints.
- The *BoolConstraint* class is a subclass of *CompositeConstraint* implementing Boolean combinations.
- The *IntVar* class is the class for all finite domain variables. It implements the following functions for generating the four kinds of events:
  – *updateInf(v:IntVar, x:integer, i:integer)* increases the lower bound of variable `v` to `x` and creates an `INCINF` event, generated by `v`'s i^{th} constraint.
  – *updateSup(v:IntVar, x:integer, i:integer)* decreases the upper bound of variable `v` to `x` and creates a `DECSUP` event, generated by `v`'s i^{th} constraint.
  – *removeVal(v:IntVar, x:integer, i:integer)* removes the value `x` from the domain of variable `v` and creates a `REMOVAL` event generated by `v`'s i^{th} constraint.
  – *instantiate(v:IntVar, x:integer, i:integer)* instantiates the variable `v` with value `x` and creates an `INSTANTIATE` event generated by `v`'s i^{th} constraint.
- The *IntDomain* class is the root class for all implementations of finite domains. It implements the following services:
  – counting the number of values stored,
  – removing a value,
  – computing/updating the lower/upper bounds of the domain,
  – iterating all values contained in the domain.
- The *IntConstraint* class is the root class for constraints that involve only finite domain variables. All classes *XXConstraint* inheriting from *IntConstraint* must implement a reaction to all four events, with the following methods (for some constraints, these four methods may call the same code):
  – *awakeOnInf(c:XXConstraint,i:integer)* for the propagation of constraint `c` upon `INCINF`(*v*) caused by *v*'s i^{th} constraint.

– *awakeOnSup(c:XXConstraint,i:integer)* for the propagation of constraint `c` upon `DECSUP`(v) caused by v's i^{th} constraint.
– *awakeOnInst(c:XXConstraint,i:integer)* for the propagation of constraint `c` upon `INSTANTIATE`(v) caused by v's i^{th} constraint.
– *awakeOnRem(c:XXConstraint,i:integer,x:integer)* for the propagation of constraint `c` upon `REMOVAL`(v,x) caused by v's i^{th} constraint.

4. Scheduling propagation events

We have introduced in section 2 the four kinds of propagation events. The `INCINF` and `DECSUP` events actually correspond to a set of smaller `REMOVAL` events. For instance, when the domain lower bound of v is raised from 2 to 4, we can either generate the event `INCINF`(v,4) or the set of events {`REMOVAL`(v,2), `REMOVAL`(v,3)}. The first case is more concise (one event instead of two), but less precise than the two other ones (the former value of the domain lower bound of v is forgotten).
Such events are introduced because some constraints are performed through bound consistency. In such cases, the propagation of one constraint yields improved bounds on variables but does not "make holes" into the domain by removing values in the middle of it. Therefore it is sufficient to reason only on bounds. It is thus a specialization over the arc consistency procedure [Mackworth 1977].
The remainder of this section will discuss issues linked to propagation event management. More precisely, we shall present the general event life-cycle, discuss scheduling policies (priorities, delays), optimization in queue management and finally, a notion of event abstraction for global constraints.

4.1. Life and death of propagation events

All events have the same lifecycle: they are first generated, then, the domain of the variable is modified, then, they are stored in an event queue, and finally, they are popped and the corresponding constraints may be waken.
Such an information flow translates into a programming pattern: below, we take the example of the `INCINF`*(v,x)* event, (the three other events have the same life cycle) and we detail the cascade of the involved function calls.

- *updateInf(v,x,idx)* — % the function generating the event.
  % *idx* is the cause of the event
  - *updateInf(v,x)* — % the function performing the update on v
    - *updateInf(v.bucket,x)* — % the function performing the update on
      % additional domain representations for v
      % stored in *v.bucket*
  - *postUpdateInf(v,idx)* — % the function posting the event in a queue
- *doAwakeOnInf(c',j)* — % later, when the event is popped from the
  % queue, all constraints *c'* linked to v are
  % asked to react to the event

A few issues can be highlighted on this programming pattern:

– In the first call (ternary *updateInf*), the third argument is the index of the constraint *c* which raised the event among all constraints linked to *v*. Passing this index as parameter allows the engine to only wake up constraints *c'* different from *c* in the last call (see section 4.4)
– At run-time, the flow can stop at the second call to *updateInf*, if the `INCINF` event does not add information to the current status of domains. This is the case when the current domain lower bound is at least as high as the newly inferred one.
– Domains may be represented by several data structures (such as bit-vectors, linked lists, interval trees, and so on) in addition to both bounds. In this case, the third call to *updateInf* performs the updates on such data structures. Such data structures may also implement explanations [Jussien, Debruyne, Boizumault 00]
– After the event has been stored (*postUpdateInf*), it is retrieved from the collection of pending events and all constraints involving the variable from the event are waken (*doAwakeOnInf).* Waking a constraint can mean either propagating it (in which case, there is one such propagation per event, à la CLP), or setting a flag on the marked constraint, such that, after all pending events have been treated, all touched constraints can be propagated (à la CSP: a constraint directed propagation loop). Thus, the programming pattern supports both style of propagation: either constraint-directed or event-directed.
– The propagation mechanism takes place only once all updates have been performed on the domain. Thus, if the event is delayed; its immediate consequences (on the domain on *v*) are taken into account before its induced consequences (through propagation). Performing the updates on the domain as early as possible is a means to increase the speed of convergence in the computation of the propagation fix point [Apt 97].

This pattern has been described on the `INCINF` event; it also applies to the three other kind of events. Note also that such a programming pattern (event generation, storage, retrieval and constraint iteration for reaction to the event) would also apply to events on other value domains (for constraints over trees, Booleans, sets, and so on).

In CHOCO, the data structures for storing events can be redefined by the user. We will show in the next sections that this pattern is generic and can be parameterized in order to support the implementation of various mechanisms such as:

– a standard AC-4 mechanism where propagation takes place in a constraint directed loop (instead of an event directed loop)
– an optimized algorithm for domain bound updates
– a layered architecture for global constraints
– and specific procedures that the user may add

4.2 Propagation event scheduling policies

The propagation phase can be described as the exploration of a graph of consequences, where each node is an event and an arc corresponds to the inference work performed by the propagation of a constraint: propagation of a constraint is triggered by an event and may, in turn, create several new propagation events. Such an inference graph may feature cycles in case there are several deduction paths to come to a same conclusion. There are (at least) two possibilities for exploring such a graph: depth first search (DFS) and breadth first search (BFS).

- DFS corresponds to a preemptive propagation of the children events: if event *e1* generates event *e2*, then event *e2* is immediately propagated without waiting for the completion of propagation of *e1*. This is easily implemented with a stack of events;
- BFS corresponds to a propagation of events by "generations". If event *e1* generates events *e2, e3* and *e4*, we wait for *e1* to be fully propagated before propagating its offspring (*e2,e3,e4*). This amounts a kind of "propagation by generations": propagating the root events first, then, their children, then their grand-children, and so on.

There is no clear argument for choosing one strategy against the other. It seems a good idea to apply DFS when an important event is raised (in order to explore immediately its consequences), while BFS supports a more fair way of waking up events. Moreover, in some tricky cases, the consequence tree can have very deep branches. An example of such a situation is displayed on the figure 5:

```
let p := makeProblem(«TRICS Tricky loop »),
    x := makeIntVar("XX",1,1000),
    y := makeIntVar("YY",1,1000),
    t := makeIntVar("TT",1,1000)  in
  (post(p, x > y),
   post(p, y > x),
   post(p, x >= 100 * T),
   post(p, y <= t),
   propagate(p) )
```

Figure 5: how many steps for discovering an infeasible problem ?

The small program above is infeasible and the call to *propagate* will detect a failure, but the chain of consequences leading to that failure has a length of 1000. We extract below two such inference paths from that tree. The first one has length 1000, while the second one is short. In such infeasible cases with deep branches, it is wiser to use BFS which will find the shortest path leading to a failure. This will not prevent long chains from being explored, but in case there exist several ways of discovering an infeasibility, it will first find the shortest one (that of least depth in the tree).

$$(x \geq 2) \overset{C_2}{\Rightarrow} (y \geq 3) \overset{C_1}{\Rightarrow} (x \geq 4) \Rightarrow \ldots \overset{C_2}{\Rightarrow} (y \geq 1001) \overset{C_1}{\Rightarrow} \bot$$

$$(t \leq 10) \overset{C_4}{\Rightarrow} (y \leq 10) \overset{C_2}{\Rightarrow} (x \leq 9) \overset{C_3}{\Rightarrow} (t = 0) \overset{C_4}{\Rightarrow} (y = 0) \overset{C_2}{\Rightarrow} (x < 0) \Rightarrow \bot$$

CHOCO's scheduling policy is the following:

- `INCINF` and `DECSUP` events are stored in a common queue and treated in a first-in first-out manner (BFS).
- `REMOVAL` events are in another fifo queue.
- `INSTANTIATE` events are stored in a stack and treated in a last-in first-out manner (DFS). They are thus popped as soon as they are pushed onto the queue. This framework gives a higher priority to instantiation compared to bound updates.

Several remarks can be made on the size of the data structures:

– the size of the stack for `INSTANTIATE` events is trivially bound by the total number n of variables.
– the size of the fifo queue for both `INCINF`, `DECSUP` events can be bound by $2n+1$. Indeed, Section 4.4 shows how one can avoid having more than one `INCINF`(v) and one `DECSUP`(v) waiting in the queue, for all variables v.
– the size of the fifo queue variables for `REMOVAL` events can unfortunately be very large (of the order of magnitude of the sum of the size of all domains).

4.3 A layered propagation architecture

Up to now, we have described the event scheduling policy at the variable level: we now describe the constraint level. Indeed, once a event variable is popped up from the queue, all the constraints in which the variable is involved are waken. When waking the constraint, we can either propagate it, or we can record that the constraint was touched and delay its propagation. The former case (immediate propagation) is well adapted for small arithmetic constraints while the latter case corresponding to the usual CSP consistency algorithms (examine constraints c one by one, and propagate those which have been touched until a fix-point is reached) is well adapted for constraints without semantics (tuple of values) and global constraints.
The overall propagation policy mixing variable-based scheduling policies with constraint-based scheduling policies is organized into layers: the reaction of constraints to events is scheduled in such manner that the quickest reactions are those of small constraints on "strong" events (events providing much information), and that reactions of "heavy" constraints (constraints with long filtering algorithms) are delayed the most. The event architecture contains the following layers:

- *level 0*: `INSTANTIATE` events are propagated immediately.
- *level 1*: `REMOVAL` events are propagated immediately thereafter.
- *level 2*: `INCINF` and `DECSUP` events are propagated afterwards, once all `INSTANTIATE` et `REMOVAL` events have been fully flushed.
- *level 3*: Once all delayed events have been treated, all constraints defined as feasible tuples of values are propagated (AC-4).
- *level 4*: Once all delayed events have been treated, all linear constraints are propagated (using a linear complexity filtering algorithm).
- *level 5*: global constraints with sub-quadratic complexity are propagated. For instance, this is the level for the complete AllDiff constraint [Régin 94]
- *level 6*: global constraints with quadratic complexity are propagated. For instance, edge-finding propagation could be placed at that level.
- *level 7*: global constraints with higher complexity are propagated.

In levels 0 to 2, only the small constraints with semantics (such as small arithmetic constraints or difference constraints) are propagated, while the other ones are delayed for upper levels. This policy amounts to achieving a set of layered fix-points: we only treat level i when levels 1 through $i-1$ have fully come to a fix-point.
For levels 3 to 7, all (delayed) constraint of the level are iterated and we test whether the constraint has been marked or not. A chained list implementation for collecting and traversing the set of marked constraints is planned for CHOCO's next release.

4.4 Optimized management of the INCINF/DECSUP event queue

This section describes a few optimizations on the management of the event queue in order to avoid generating redundant events or waking up constraints that will not yield new information.
The constraint queue stores events as triplets *(evt, v, idx)*, where *evt* is the event (`INCINF` or `DECSUP`), *v* is the domain variable that has been modified and *idx* represent the cause of the event (the index of the constraint that generated the event among all constraints linked to *v*). In some cases detailed at the end of section 4.4, the cause of the event is deliberately forgotten, and *idx* = 0.

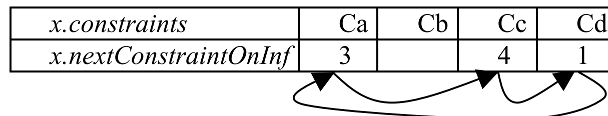
Selecting the constraints to wake up

When an event *(*`INCINF`*, x, i)* is popped from the queue, the constraints connected to *v* are iterated in order to react to the event. The simplest way to do so would be to iterate over the list *x.constraints* and call for all k, $1 \leq k \leq n$ (with $n = size(x.constraints)$), *doAwakeOnInf(x.constraints[k], x.indices[k])*. The second parameter (*v.indices[k]*) is the index of *x* in its kth constraint. We add two optimizations over this simple iteration, in order to avoid useless wakes:

1. It is useless to have the very cause of the event react to the event. Therefore, in the above loop, one should not consider the value *k=i.* In order to do so, we replace the iteration for *k* from 1 to *n* by an iteration from *i+1* to *n* and from *1* to *i-1* (note that this is still valid in the case *i=0*, which is our aforementioned convention when the cause of the event is ignored).
2. Other values can be skipped by *k* during this iteration. A first reason is that some constraints will never provide us with new information upon `INCINF` events to some of their variables (the constraint *x*>*y* should not be woken upon `INCINF`(*x*) or `DECSUP`(*y*) events). A second reason is that some constraints that are fully satisfied before being fully instantiated (which is the case for *x*>*y* when *x.inf*>*y.sup*) should not be woken. In both cases, we want the iteration to skip the index for *k*.

In order to perform this restricted iteration, CHOCO implements a list of pointers (the field *x.nextConstOnInf* for the variable *x)* containing, for each index *i*, the index of the next constraint that should be woken after constraint at index *i.*
The figure below depicts a situation where *x* is linked to four constraints *Ca, Cb, Cc,* and *Cd*, among which only *Ca, Cb* and *Cd* are active on the `INCINF`(*x*) event.

x.constraints	Ca	Cb	Cc	Cd
x.nextConstraintOnInf	3		4	1



In this example, when the event (`INCINF`*, x, 3*) occurs, we know that *Cc* is the cause of the event (it is the 3rd constraint on *x)*, and we can perform the iteration starting with *k=x.nextConstOnInf[3]=4*, therefore, asking *Cd=x.constraints[4]* to react to the update of *x.inf*, then going on with *k=x.nextConstOnInf[4]=1*, therefore, asking *Ca=x.constraints[1]* to react to the update of *x.inf,* and stopping the iteration thereafter because *k=x.nextConstOnInf[1]=3* brings us back to the start of our iteration (the index of the cause of the event).

Removing redundant events

When a new event is added to the queue, we check that no similar event is already present[4] in the queue: indeed, if an event *e1*=(INCINF, *x, i*) is already present in the queue, and if we add *e2*=(INCINF, *x, j*), the reaction of the constraints to *e2* will be virtually redundant with the reactions to *e1*. Indeed, the reaction to *e1* will take place after the two updates on the value of *x.inf*. It is thus useless to react a second time (for *e2)* to the increase of the lower bound of *x*. Therefore, the second event *e2* is not added into the queue, since its propagation redundant with that of *e1*.

The situation gets trickier when both optimizations (not waking up the cause of an event and removing redundant events) are jointly performed. In case the events have the same origin (*i=j),* the redundant event (*e2)* can be simply forgotten. Otherwise, when the events have different origins ($i \neq j$), it is no longer correct to apply both optimizations, that is to ignore *e2* and to react globally to e1 and e2 by waking up all constraints linked to *x* but the ith (which caused *e1*).

Instead, in order to dismiss *e2* from the queue of pending events, we forget the cause of event *e1* (replacing *i* by 0) and iterate all constraints when popping *e1:* among them, waking up the ith is motivated only by *e2*, waking up the jth is motivated only by *e1* and waking up the others is motivated both by *e1* and *e2*.

4.5 Strenghtening events

There are a few tricky cases, where events get transformed. This is due to the redundant representations of the domain. For instance, suppose that the domain of *x* is {1, 4, 7, 9}:

- A REMOVAL event ($x \neq 1$) will reduce the domain to {4,7,9}. In the interval approximation of the domain of the variable *x*, this event does not amount to the removal of one but several values. Therefore, we could generate a INCINF event instead of the REMOVAL event.
- An event ($x \geq 8$) will reduce the domain to {9}. Therefore, what appeared to be an INCINF event, by only looking at the interval approximation of the domains should be an INSTANTIATE event, yielding stronger propagation.

Therefore, in the flow of function calls presented above, *updateInf(x,a,idx)* will not always generate a call to *postUpdateInf(x,idx)*, but sometimes to *postInstantiate(x,j)*. A tricky question is whether, in those cases of event strengthening (transforming an INCINF event into an INSTANTIATE event), the *idx* parameter representing the cause of the event generation should be kept. In fact, no. If a constraint generated an update $x \geq 8$, and if this translates into the instantiation $x=9$, then that constraint should be informed about the additional information. Therefore, we should generate a call to *postInstantiate(x,idx)*. We actually rather generate a call to *postInstantiate(x,0)*, thus forgetting the cause of the instantiation; indeed, the constraint *c=x.constraints[idx]* that generated the event $x \geq 8$ could perform additional inferences if it knew that this translated into $x=9$. Therefore, in the iteration of all constraints on *x, c* should not be skipped. In order to ensure that *c* will not be skipped (first optimization of section 4.4)

[4] In order to know whether *e1* is present in the queue, the variable *x* records in two fields the address of events INCINF(*x*) and DECSUP(*x*) if they are present in the queue.

and will be asked to react to the strengthened event $x=9$, we need to replace the second parameter of *postInstantiate* by 0.

4.6. Abstracting events

The last case of event transformation takes place with global constraint propagation: instead of strengthening the event with information from state of the domain, global constraint forget part of the details of the events, they abstract the event in some respect.

Since the propagation of global constraints is delayed (in the levels 4 and above described in section 4.3), global constraints are often asked to react after several events have taken place. For the sake of efficiency, if *k* events have happened since the last wake up of the global constraints, there will be less than *k* procedures calls, because the wake up algorithm will consider several of those events at a time, taking a gross view of what has changed. Such a way of forgetting the details of the exact propagation events is an abstraction mechanism.

The simplest way of doing so for a global constraint is to abstract all possible events into one abstract event ("something has changed"). In this case, the global constraint implements only one reaction mechanism, independent from the amount of changes in the domains since the last time that the global constraint was woken. This is the actual procedure for constraints defined as tuples of values (level 3).

A slightly more subtle case corresponds to the case of linear constraints where two abstract events are considered. Consider the general linear constraint

$$\sum_i a_i X_i - \sum_j b_j Y_j + c = 0$$

where all coefficients a_i and b_j are positive. Let us introduce two bounds:

$$lb = \sum_i a_i X_i.inf - \sum_j b_j Y_j.sup + c$$

$$ub = \sum_i a_i X_i.sup - \sum_j b_j Y_j.inf + c$$

There are two propagation rules for such a constraint [Harvey 98]:

$$\forall i_0,\quad X_{i0}.sup - ub/a_{i0} \le X_{i0} \le X_{i0}.inf - lb/a_{i0}$$

$$\forall j_0,\quad Y_{j0}.sup - lb/b_{j0} \le Y_{j0} \le Y_{j0}.inf - ub/b_{j0}$$

Therefore when any event occurs on variables *X* and *Y*, it is sufficient to note whether the bound *lb* or *ub* has been modified. Therefore all events INCINF(X_i), DECSUP(X_i), INCINF(Y_j), DECSUP(Y_j) are abstracted in two global events: one when *lb* is improved, and one when *ub* is improved. The propagation loop is very simple:

- when *lb* has been improved, all $X_i.sup$ and all $Y_j.inf$ are updated
- when *ub* has been improved, all $X_i.inf$ and all $Y_j.sup$ are updated

Each global constraint has its own abstract events. For instance binary matching constraints propagated with the Hungarian method [Caseau, Laburthe 97] have two different events: whether edges in the optimal matching have been removed and whether edges outside this optimal matching have been removed.

5. Conclusion

In this paper, we have introduced CHOCO, a Constraint Programming library and we have presented its implementation. There are two contributions in the paper. The first contribution is an object-oriented architecture that is modular enough to easily support extensions. It defines a notion of internal API dedicated to CP with classes and low-level services. It took us some time and a few dead ends to find it, so we hope the paper will be of help to future CP system implementers. The second contribution is the management of propagation events, a generic framework which has been optimised for handling different kinds of constraints, with or without semantics, and with global constraints. To our knowledge, practical propagation event management policies applying to small and global constraints have not been published yet. This is ongoing work, and a future paper will assess the efficiency of some of CHOCO's implementation options with comparisons and benchmarks.

6. References

A. Aggoun, N. Beldiceanu, *Extending the CHIP compiler to solve complex scheduling problems*, Journal of Mathematical and Computer Modelling, 17(7), 57-63, Pergamon Press, 1993.

A. Aggoun, N. Beldiceanu, *Overview of the CHIP compiler system*, Proceedings of the 8th International Conference on Logic Programming, 775-789, MIT Press, 1991.

K.R. Apt, *From Chaotic Iteration to Constraint Propagation*, Proc. of the 24th International Colloquium on Automata, Languages and Programming (ICALP'97), Lecture Notes in Computer Science 1256, p. 36-55, Springer, 1997.

Y. Caseau, F. Laburthe, *Introduction to the CLAIRE programming language*, rapport technique du LIENS 96-9, Ecole Normale Supérieure, 1996.
see http://www.ens.fr/~caseau/claire.html or http://www.clairelanguage.com

Y. Caseau, F. Laburthe, *Solving Various Matching Problems with Constraints*, Proc. of the 3rd International Conference on Principles and Practice of Constraint Programming, CP'97, G. Smolka ed., Lecture Notes in Computer Science 1330, Springer, p. 17-31, 1997.

W. Harvey, P. Stuckey, *Constraint Representation for Propagation,* Proc. of the 4th International Conference on Principles and Practice of Constraint Programming, CP'98, M. Maher, J.-F; Puget eds., Lecture Notes in Computer Science 1520, Springer, p. 235-249, 1998.

N. Jussien 2000, R. Debruyne, P. Boizumault, *Maintaining arc-consistency within dynamic backtracking,* Proceedings of CP'2000.

J.-L. Laurière *A Language and a Program for Stating and Solving Combinatorial Problems*, AI 10, 29-127, 1978.

A.K. Mackworth *Consistency in Networks of Relations,* AI Journal, 8(1), p.99-118, 1977.

J.-F. Puget. *Programmation par contraintes orientée objet*. Douzièmes journées internationales sur les systèmes experts et leurs applications, Avignon, France, 1992

J.C. Régin, *A filtering Algorithm for constraints of difference in CSP,* Proc. of AAAI'94, Seattle, p. 362-367, 1994.