Comet in Context

Laurent Michel

University of Connecticut,
Storrs, CT 06269-3155
ldm@engr.uconn.edu

Pascal Van Hentenryck

Brown University, Box 1910,
Providence RI 02912
pvh@cs.brown.edu

ABSTRACT

Combinatorial optimization problems naturally arise in many application areas, including logistics, manufacturing, supply-chain management, and resource allocation. They often give rise to complex and intricate programs, because of their inherent computational and software complexity. There is thus a strong need for software tools which would decrease the distance between the specification and the final program.

This paper contains a brief description of COMET, an object-oriented language supporting a constraint-based architecture for neighborhood search. It contrasts COMET to constraint programming languages and shows how constraint programming and COMET provides many of the same benefits for constraint satisfaction and neighborhood search respectively. In particular, COMET supports a layered architecture cleanly separating modeling and search aspects of the programs, constraints encapsulating incremental algorithms, and various control abstraction to simplify neighborhood exploration and meta-heuristics.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries,Reuse models*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Constraints, Frameworks*

General Terms

Algorithms,Experimentation,Languages

Keywords

Combinatorial optimization, constraint programming, neighborhood search, incremental algorithms

1. INTRODUCTION

Combinatorial optimization problems are ubiquitous in many parts of our societies. From the airline industry to

courier services, from supply-chain management to manufacturing, and from facility location to resource allocation, many important decisions are taken by optimization software every day. In general, optimization problems are extremely challenging both from a computational and software engineering standpoint. They cannot be solved exactly in polynomial time and they require expertise in algorithmics, applied mathematics, and the application domain. Moreover, the resulting software is often large, complex, and intricate, which makes it complicated to design, implement, and maintain. This is very ironic, since many optimization problems can be specified concisely. The distance between the specification and the final program is thus considerable, which indicates that software tools are seriously lacking in expressiveness and abstractions in this application area.

Because of the nature of optimization problems, no single approach is likely to be effective on all problems, or even on all instances of a single problem. It is thus of primary importance that all major approaches to optimization be supported by high-level tools automating many of the tedious and complex aspects of these applications. Historically, most of the research has focused on constraint and mathematical programming, which are now supported by a rich variety of modeling and programming tools (e.g., [11, 20, 46, 27, 49, 12]). In contrast, neighborhood search, one of the oldest optimization techniques, has been largely ignored until recently (e.g., [10, 44, 31, 54]). This is a serious gap in the repertoire of tools for optimization. This limitation is further exarcerbated by the fact that solving optimization problems remains a very experimental endeavour: what will, or will not, work in practice is hard to predict. Proper software tools facilitate this experimentation and often result in higher quality solution techniques, since users are more likely to try out various avenues.

At a conceptual level, neighborhood search explores a graph where nodes represent solutions (or configurations) and arcs represent transitions from a solution to a neighboring solution. How to define this neighborhood graph and how to explore it effectively are fundamental issues which have received considerable attention in recent years. Neighborhood search is the technique of choice for a variety of fundamental applications. For instance, at the time of writing, the best approach to the travelling tournament problem, an abstraction of Major League Baseball scheduling, is a neighborhood search method which significantly outperforms constraint and mathematical programming. The same can be said of many important problems, such as vehicle routing, frequency allocation, and many resource alloca-

tion and scheduling problems. Equally important perhaps is the belief that hybrid algorithms, which combines several approaches in innovative ways, are likely to produce the next level of improvements in that area. Recent results in routing and scheduling have indicated the promises of hybridization. Yet neighborhood search algorithms are weakly supported in modeling and programming tools.

The COMET project, which emerged from earlier research on LOCALIZER, was initiated to address these needs. COMET is an object-oriented language supporting a constraint-based architecture for neighborhood search and featuring novel declarative and control abstractions. COMET decreases the size of neighborhood search programs significantly and enhances compositionaly, modularity, and reuse for this class of applications.

From a language standpoint, COMET drew much inspiration from constraint programming. It features both a layered architecture with modeling and search components, and a rich constraint vocabulary, which includes numeric and structural constraints. Its control abstractions depart from typical constraint programming languages however, since the issues raised by neighborhood search, i.e., graph exploration, differ in nature from those of systematic search, which is mostly concerned by tree searching. In particular, first-class closures is a common theme in these abstractions and make it possible to achieve elegance and simplicity, while not sacrificing efficiency.

From a computational model standpoint, COMET significantly differ from existing constraint programming systems, which is natural since it supports a very distinct class of applications. In COMET, constraints are not used to prune the search space. Rather they maintain properties that are then used to direct the graph exploration effectively. For instance, a constraint typically maintains how much its variables contribute to its violations and this information is often useful to choose the next neighbor. In addition, constraints are *differentiable objects*, which means that they can be queried to determine the impact of local moves on their properties. At an implementation level, constraints, and differentiable objects in general, encapsulate efficient incremental algorithms which arise in many applications. However, at a conceptual level, there are some nice similarities between COMET and traditional constraint programming systems. In particular, both of them cleanly separate the problem modeling from the search in the source program, although the execution interleaves these components in complex ways. As a consequence, they provide attractive modularity, compositionality, and extensibility, which are critical in this application area.

This paper presents a brief overview of COMET from a constraint programming perspective, because it gives us a wonderful opportunity to acknowledge the scientific vision of Paris Kanellakis. As early as 1994, Paris was gently encouraging us to study how constraint programming languages could accommodate neighborhood search. Paris was actively involved in Constraint Query Languages (CQL) [23] and functional programming [22] at the time (among many other topics) and he had made, earlier in his career, fundamental contributions to neighborhood search [24], so he could somehow "foresee" some of the synergies described in this paper. Our first contribution in this area was the LOCALIZER system mentioned earlier. But it is only recently that we realized how much a constraint and functional programming
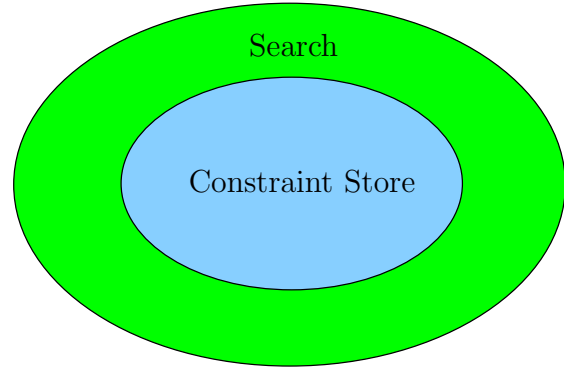

Figure 1: The CP Architecture

perspective brings to neighborhood search through differentiable objects [32] and first-class closures [52]. The seeds planted by Paris in 1994 have now given us some exciting research avenues, which we are gladly pursuing.

The rest of this paper is organized as follows. Section 2 briefly introduces constraint programming and its application to combinatorial optimization, which is the most widely-used branch of constraint programming at this point. Section 3 informally describes the constraint-based architecture for neighborhood search of COMET. Section 4 concludes the paper by contrasting these two approaches at various levels, highlighting their similarities and their differences.

2. CONSTRAINT PROGRAMMING

Constraint programming is a recent entry to the field of programming languages. Its essence is a two-level architecture integrating a *constraint* and a *programming* component as shown in Figure 1. The constraint component, often called the constraint store, provides the basic operations of the architecture and consists of a reasoning system about fundamental properties of constraint systems. The constraint store contains the constraints accumulated at some computation step and supports various queries and operations over them. Operating around the constraint store is a programming-language component that specifies how to combine the basic operations, often in nondeterministic ways, since search is so fundamental in many applications of constraint programming.

The constraint-programming framework has been applied to many areas, including computer graphics (e.g., [5]), software engineering (e.g., test generation in [14]), databases (e.g., [23]), hybrid systems (e.g., [26, 16]), finance (e.g., [3, 19]), engineering (e.g., [15, 18]), circuit design (e.g., [45]), computational molecular biology (e.g., [4]) and, of course, combinatorial optimization. Given the diversity of these application areas, it is not surprising that the programming and constraint components can be of fundamentally different nature. However, when restricting attention to combinatorial optimization, constraint programming systems are generally based on a common set of design principles that stem from their roots in Constraint Logic Programming (CLP) ([7, 8, 21, 47]). More precisely, the constraint programming approach to combinatorial optimization can be characterized (at this point) by two main features:

1. an expressive language to state combinatorial opti-

mization problems, including a rich constraint language and the ability to specify search procedures;

2. a new computational model to solve combinatorial optimization problems which focuses on using constraints, or feasibility information, to reduce the search space.

This section reviews these two features independently and restricts attention to combinatorial optimization only.

2.1 The Language

Solving a combinatorial-optimization problem in constraint programming amounts to describing the problem constraints and to specifying a search procedure.

Constraints in constraint programming are generally expressed in a rich language that includes linear and nonlinear constraints, the ability to index arrays with variables, logical combinations of constraints, cardinality constraints, and higher-order constraints, as well as structural, or "global", constraints. Global constraints are a natural way to integrate many algorithms from theoretical computer science and operations research. They capture a substructure that arises in many applications and is amenable to efficient pruning. In addition, some constraint languages also offer set variables and set constraints.

The ability to specify a search procedure is another fundamental feature of constraint-programming languages. This aspect of constraint programming was present since its inception (e.g., [51, 8, 47]) and was considered critical to obtain reasonable efficiency on difficult combinatorial problems. Much research in recent years was devoted to language support for search procedure (e.g., [27, 48, 49]). Very high-level nondeterministic constructs are now available to specify the search tree to explore in elegant and concise ways. Simultaneously, the language Oz [41] pioneered novel features to program search strategies (e.g., limited discrepancy search [17]), which specifies how to explore the search tree. Support for search strategies are now available in several modern languages [37, 53].

It is also important to stress that the constraint programming framework is essentially language-independent. Early constraint languages were based on logic programming. Object-oriented libraries have been very successful in bringing constraint programming to industry, while modeling languages now address the need of mathematical modelers.

2.2 The Computation Model

Besides its programming language contributions, constraint programming has also contributed a novel way of approaching the solving of combinatorial optimization problems, which focuses mostly on exploiting feasibility information. The computational model of constraint programming is a combination of two processes:

1. a constraint satisfaction engine which uses constraints *independently* (or *locally*) to reduce the domains of the variables.

2. a search engine which decomposes a problem into subproblems when constraint satisfaction cannot reduce the search space.

The search element of the computational model is, in essence, similar to the branch-and-bound approach of integer programming. The main originality in the computational model
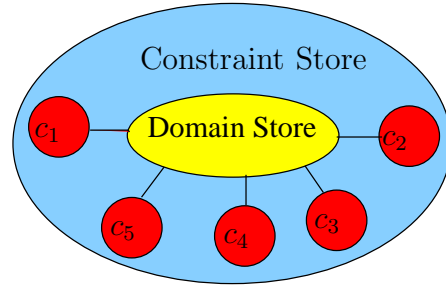

Figure 2: The CP Constraint Store

of constraint programming lies in the underlying constraint-satisfaction algorithms or, more generally, in the constraint-store organization.

Figure 2 depicts the architecture of the constraint store. The core of the architecture is the *domain store* that associates a domain with each variable. The domain of a variable represents its set of possible values at a given computation state (e.g., at a node in the tree search). Gravitating around these domains are the various constraints, each of which has no knowledge of the other constraints. Associated with each constraint is a constraint-satisfaction algorithm whose primary role is to perform two main tasks:

1. to determine if the constraint is consistent with the domain store, i.e., if there exist values for the variables in their domains that satisfy the constraint;

2. to apply a filtering algorithm to remove, from the domains of the constraint variables, values that do not appear in any of its solutions.

In addition, the constraint-satisfaction algorithm may add new constraints to the constraint store as we discuss later in the paper.

The constraint solver can then be viewed as a simple iterative algorithm whose basic step consists of selecting a constraint and applying its constraint-satisfaction algorithm. The algorithm terminates when a constraint is inconsistent with respect to the domain store or when no more domain reductions are possible. Note that, on termination, there is no guarantee that there exists a solution to the constraint store. The constraints may all have a local solution with respect to the domain store but these solutions may be incompatible globally. This architecture, which is now the cornerstone of most modern constraint-programming systems, was pioneered by the CHIP system [11, 47, 51]) which included a solver for discrete finite domains based on constraint-satisfaction techniques (e.g., [29, 34, 56]).

It is important to emphasize that, unlike integer and linear programming, constraint-programming systems may support *arbitrarily complex constraints*. These constraints are not restricted to linear constraints, or even nonlinear constraints, and they may represent complex relations between their variables. For instance, a constraint may require that all its variables be assigned distinct values or that a set of activities do not overlap in time. As a consequence, it is useful, and correct, to think of a constraint as representing *an interesting subproblem* of the application and it is one of the fundamental issues in constraint programming to isolate classes

```
int n = 512;
range Size = 1..n;
var Size queen[Size];
int neg[i in Size] = -i;
int pos[i in Size] = i;

solve {
   forall(i in Domain) {
      alldifferent(queen);
      alldifferent(queen,neg);
      alldifferent(queen,pos);
   }
};
search {
   forall(i in Size
         ordered by increasing dsize(queen[i]))
      tryall(v in Size);
         queen[i] = v;
};
```

Figure 3: A Simple N-Queens Constraint Program

of constraints that are widely applicable and amenable to efficient implementation.

It is also important to stress that, ideally, the constraint-satisfaction algorithm associated with a constraint should be complete (i.e., it should remove all inconsistent values) and run in polynomial time. Such complete algorithms enforce arc consistency [29]. However, enforcing arc consistency may sometimes prove too hard (i.e., it may be an NP-hard problem), in which case simpler consistency notions are defined and implemented. This is the case, for instance, in scheduling algorithms.

2.3   A Simple Example

We now illustrate constraint programming on a simple example, which we will also use for neighborhood search later on. The n-queens problem consists of placing n queens on a chessboard of size $n \times n$ so that no two queens lie on the same row, column, or diagonal. Since no two queens can be placed on the same column, a simple program of this problem consists of associating a queen with each column and searching for an assignment of rows to the queens so that no two queens are placed on the same row or on the same diagonals. Figure 3 depicts a program for the n-queens problem in the modeling language OPL.

The OPL program illustrates the structure typically found in constraint programs: the declaration of the data, the declaration of decision variables, the statement of constraints, and the search procedure, as will be illustrated shortly. The program first declares an integer n, and a range Size. It then declares an array of n variables, all of which take their values in the range 1..n. In particular, variable queen[i] represents the row assigned to the queen placed in column i. The next two instructions declare arrays of constants which will be used to state the constraints.

The solve instruction defines the problem constraints, i.e., that no two queens should attack each other. It indicates that the purpose of this program is to find an assignment of values to the decision variables that satisfies all constraints. The basic idea in this program is to generate, for all $1 \le i < j \le n$, the constraints
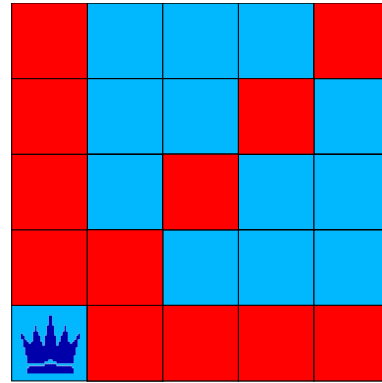

Figure 4: The 5-Queens Problem After One Choice

```
queen[i] <> queen[j]
queen[i] - i <> queen[j] - j
queen[i] + i <> queen[j] + j
```

where the symbol <> means "not-equal." This is achieved by the "global" alldifferent constraint [38] with proper offsets. As mentioned earlier, structural constraints of this type are critical in large and complex applications, since they encapsulate efficient pruning algorithms for substructures arising in many applications.

The rest of the program specifies the search procedure. Its basic idea is to consider each decision variable in sequence and to generate, in a nondeterministic way, a value for each of them. If, at some computation stage, a failure occurs (i.e., the domain of a variable becomes empty), the implementation backtracks and tries another value for the last queen assigned. More precisely, the forall instruction corresponds to an ordered *and-node* in artificial intelligence terminology and executes its body for each value i in Size. The tryall instruction corresponds to an ordered *or-node* in artificial intelligence terminology and is nondeterministic, i.e., it specifies a choice point with a number of alternatives and one of them must be selected. In the context of the queens problem, each of these alternatives corresponds to the assignment of a row to a queen. Note that the forall instruction features a dynamic ordering of the variables: at each iteration, it selects the variable with the smallest domain, implementing the so-called first-fail principle.

To illustrate the computational model, consider the five-queens problem. Constraint propagation does not reduce the domains of the variables initially. OPL thus generates a value, say 1, for one of its variables, say queen[1]. After this nondeterministic assignment, constraint propagation removes inconsistent values from the domains of the other variables, as depicted in Figure 4. The next step of the generation process tries the value 3 for queen[2]. OPL then removes inconsistent values from the domains of the remaining queens (see Figure 5). Since only one value remains for queen[3] and queen[4], these values are immediately assigned to these variables and, after more constraint propagation, OPL assigns the value 4 to queen[5]. A solution to the five-queens problem is thus found with two choices and without backtracking, i.e., without reconsidering any of the choices.
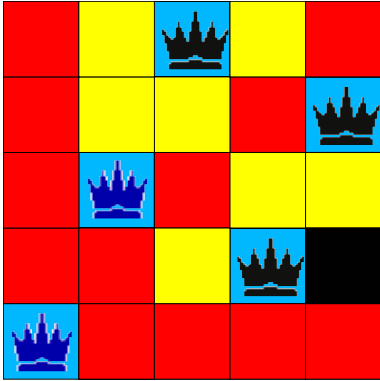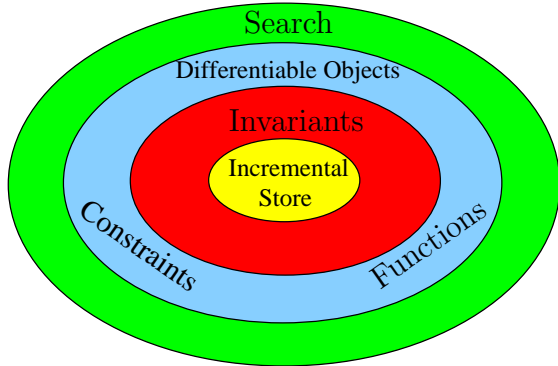
Figure 5: The 5-Queens Problem After Two Choices

Figure 6: The Comet Architecture.

It is important to stress the clean separation between the constraints and the search procedure description in this program. Although these two components are *physically separated* in the program, they *cooperate* in solving the problem. Whenever a variable is assigned a value, a constraint propagation step is initiated, which prunes the search space. This pruning in turn affects the control behaviour through backtracking or through the heuristics.

3. NEIGHBORHOOD SEARCH

We now turn to COMET and we show that much of the style, and many of the benefits, of constraint programming carry over to neighborhood search. In particular, we show that COMET also implements a layered architecture, cleanly separates modeling and search, and supports a rich language for constraints and search procedures. The technical details and the computational model significantly differ however, due to the nature of the underlying approach to combinatorial optimization.

3.1 The Architecture

The architecture is shown in Figure 6. It consists of a declarative and a search component organized in three layers, which we now review.

3.1.1 Invariants

The kernel of the architecture is the concept of *invariants* (or one-way constraints) over algebraic and set expressions [30]. Invariants are expressed in terms of incremental variables and specify a relation which must be maintained under assignments of new values to its variables. For instance, the code fragment

```
inc{int} s(m) <- sum(i in 1..10) a[i];
```

declares an incremental variable `s` of type int (in a solver `m`) and an invariant specifying that `s` is always the summation of `a[1],...,a[10]`. Each time, a new value is assigned to an element `a[i]`, the value of `s` is updated accordingly (in constant time). Note that the invariant specifies the relation to be maintained incrementally, not how to update it. Incremental variables are always associated with a local solver (`m` in this case). This makes it possible to use a very efficient implementation by dynamically determining a topological order in which to update the invariants. As we will see later, COMET supports a wide variety of algebraic, graph, and set invariants.

3.1.2 Differentiable Objects

Once invariants are available, it becomes natural to support the concept of *differentiable objects*, a fundamental abstraction for local search programming. *Differentiable objects maintain a number of properties (using invariants or graph algorithms) and can be queried to evaluate the effect of local moves on these properties.* They are fundamental because many neighborhood search algorithms evaluate the effect of various moves before selecting the neighbor to visit. Two important classes of differentiable objects are constraints and functions. A differentiable constraint maintains properties such as its satisfiability, its violation degree, and how much each of its underlying variables contribute to the violations. It can be queried to evaluate the effect of local moves (e.g., assignments and swaps) on these properties. A differentiable function maintains the value of a (possibly) complex function and can also be queried to evaluate the variation in the function value under local changes.

Differentiable objects capture combinatorial substructures arising in many applications and are appealing for two main reasons. On the one hand, they are high-level modeling tools which can be composed naturally to build complex neighborhood search algorithms. As such, they bring into neighborhood search some of the nice properties of modern constraint satisfaction systems. On the other hand, they are amenable to efficient incremental algorithms that exploit their combinatorial properties. The use of combinatorial constraints is implicitly emerging from a number of research projects: It was mentioned in [55] as future research and used, for instance, in [6, 13, 35] as building blocks for satisfiability problems. Combinatorial functions play a similar role for optimization problems, where it is important to evaluate the variation of complex objective functions efficiently.

The `AllDifferent` constraint, which we encountered earlier, is an example of differential object. In its simplest form, `AllDifferent(a)` receives an array `a` of incremental variables and holds if all its variables are given different values. The `AllDifferent` constraint maintains a variety of properties incrementally. They include its violation degree, i.e., how many constraints $a[i] \neq a[j]$ are violated, as well

```
abstract class Constraint {
  inc{int} true();
  inc{int} violationDegree();
  inc{int} violations(inc{int} var);
  int getAssignDelta(inc{int} var,int val);
  int getSwapDelta(inc{int} v,inc{int} w);
  ...
}
```

Figure 7: The Abstract Class `Constraint`

as the set of variables which occur in such violations. Observe that the same structural constraint, which is natural in many applications, is being used in two very different ways: to prune the search space in constraint programming and to maintain incremental properties in COMET.

In COMET, differentiable constraints are objects implementing the abstract class `Constraint`, an excerpt of which is shown in Figure 7. The first three methods give access to incremental variables that maintain important properties of the constraint: its satisfiability, its violation degree, and the violations of its variables. The other two methods shown in Figure 7 are particularly interesting as they evaluate the effect of assignments and swaps on the violation degree of the constraint. Method `getAssignDelta(var,val)` returns the variation of the violation degree when variable `var` is assigned value `val`. Similarly, method `getSwapDelta(v,w)` returns the variation of the violation degree when the values of variables `v` and `w` are swapped. Although this paper presents several differentiable constraints available in COMET, observe that COMET is an open language where users can add their own constraints by subclassing `Constraint`. (Similar considerations apply to differentiable functions.)

Constraints can be composed naturally using *Constraints systems*. A constraint system groups a collection of constraints together and maintains the satisfiability and violation degree of the set incrementally. The violation degree of a set of constraints is the summation of the violation degrees of its constraints. Being constraints themselves, constraint systems are differentiable and can also be queried to evaluate the effect of local moves. Constraint systems make neighborhood search algorithms more *compositional* and easier to extend. In particular, they allow new constraints to be added in the declarative component of an algorithm without changing the search component.

These two layers, invariants and differentiable objects, constitute the declarative component of the architecture.

3.1.3 Control Abstractions

The third layer of the architecture is the search component which aims at simplifying the neighborhood exploration and the implementation of heuristics and meta-heuristics, two other critical aspects of neighborhood search algorithms. The search component features high-level constructs and abstractions to foster and increase separation of concerns. It features multidimensional selectors, since selection is generally quite involved in neighborhood search, as well as a variety of abstractions to express the neighborhood exploration concisely and to control the graph exploration. They include events, the `neighbor` constructs, as well as checkpoints, which simplify the implementation of variable-depth neighborhood search.

One of the main issues addressed by these abstractions is

the temporal disconnection between the definition of a behavior and its use which typically plagues the implementation of neighborhood search algorithms. This issue arises in meta-heuristics and in neighborhood exploration for applications whose neighborhoods are heterogenous and consists of various types of moves (e.g., [28, 25, 9, 2]). COMET abstractions heavily rely on first-class closures to address this temporal disconnection and implements events, neighbors, and checkpoints.

3.2 The Queens Problem

```
int n = 512;
range Size = 1..n;
LocalSolver m();
UniformDistribution distr(Size);

inc{int} queen[i in Size](m,Size) := distr.get();
int neg[i in Size] = -i;
int pos[i in Size] = i;

ConstraintSystem S(m);
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen,neg));
S.post(new AllDifferent(queen,pos));
inc{set{int}} conflicts(m);
conflicts <- argMax(q in Size) S.violations(queen[q]);
m.close();

while (S.violationDegree())
  select(q in conflicts)
    selectMin(v in Size)(S.getAssignDelta(queen[q],v))
      queen[q] := v;
```

Figure 8: The Queens Problem in Comet.

We now reconsider the queens problem for neighborhood search. Figure 8 depicts a COMET program which shares many features with the earlier program, although the two programs implement fundamentally different algorithms. The algorithm in Figure 8 implements the min-conflict heuristic [33]. It starts with an initial random configuration. Then, at each iteration, it chooses the queen violating the largest number of constraints and moves it to a position minimizing its violations. This step is iterated until a solution is found.

Once again, the COMET program associates a queen with every column and it uses `queen[i]` to denote the row of the queen on column `i`. As before, the program starts by declaring the size of the board and a range. It then declares a local solver, which will hold the incremental variables, the invariants, and the constraints, and a uniform distribution. The next instruction

```
inc{int} queen[i in Size](m,Size) := distr.get();
```

declares the incremental variables and initializes them randomly. Note that each such variable receives, in its constructor, the local solver and, in this case, its range of values. Incremental variables are central in COMET. They are used in invariants and constraints, and changes to their values trigger events and induce a propagation step that updates all invariants and constraints directly or indirectly affected by the changes. Note the use of the assignment operator `:=`, which assigns a value of type T to an incremental variable of type `inc{T}`. By contrast, the operator `=` assigns references. The following two instructions declare the traditional arrays for storing the offsets.

The next block of instructions is particularly interesting and describes the declarative or modeling component of the application. Instruction

```
ConstraintSystem S(m);
```

declares a constraint system S, while the instructions

```
S.post(new AllDifferent(queen));
S.post(new AllDifferent(queen,neg));
S.post(new AllDifferent(queen,pos));
```

add the problem constraints to S. Observe that these are the very same constraints as in the constraint programming solution. The final instructions of the declarative component

```
inc{set{int}} conflicts(m);
conflicts <- argMax(q in Size) S.violations(queen[q]);
```

are particularly interesting. The first instruction declares an incremental variable conflicts whose values are of type "set of integers". The second instruction imposes an invariant ensuring that the values of conflicts always be the set of queens that violate the most constraints. Observe that S.violations(queen[q]) returns an incremental variable representing the number of violations in S. Operator argMax(i in S) E simply returns the set of values v in S which maximizes E. The last instruction

```
m.close()
```

closes the model, which enables COMET to construct a dependency graph to update the constraints and invariants under changes to the incremental variables.

Observe that the declarative component only specifies the properties of the solutions, as well as the data structures to maintain. It does not specify how to update the constraints, e.g., the violations of the variables, or how to update the conflict set. These are performed by the COMET runtime system, which uses optimal incremental algorithms in this case.
The final part of the program

```
while (S.violationDegree())
   select(q in conflicts)
      selectMin(v in Size)(S.getAssignDelta(queen[q],v))
         queen[q] := v;
```

states the search strategy. It iterates until the violation degree of the system is zero, meaning that all constraints are satisfied. Each iteration randomly selects a queen q in the conflict set, selects a value v for queen q that minimizes the number of violations, and assigns v to queen[q]. Note that the target row selection carried out by

```
selectMin(v in Size)(S.getAssignDelta(queen[q],v))
```

uses the ability to query the constraint system to find out the impact of assigning v to queen q. This query can be performed in constant time, thanks to the invariants maintained in each of the constraints.

There are a couple of observations to make at this point. First, observe that the search and declarative components are clearly separated in the program, as was the case with the constraint programming solution. It is thus easy to modify one of them without affecting the other. For instance, it is possible to add new constraints to the constraint system

without any change to the search component. Similarly, it is easy to replace the search component without affecting the declarative component. Although the two components are physically separated in the program code, they closely collaborate during execution. The search component uses the declarative component to guide the search, while the assignment queen[q] := v starts a propagation phase which updates all invariants and constraints. Once again, these are the same appealing features as constraint programming. Finally, observe the high-level nature of the resulting program. The declarative component is close to a formal specification of the problem, while the search component is a very abstract description of the search strategy. In fact, it can be observed that this search component is not at all problem-specific and could almost be reused as such in other contexts.
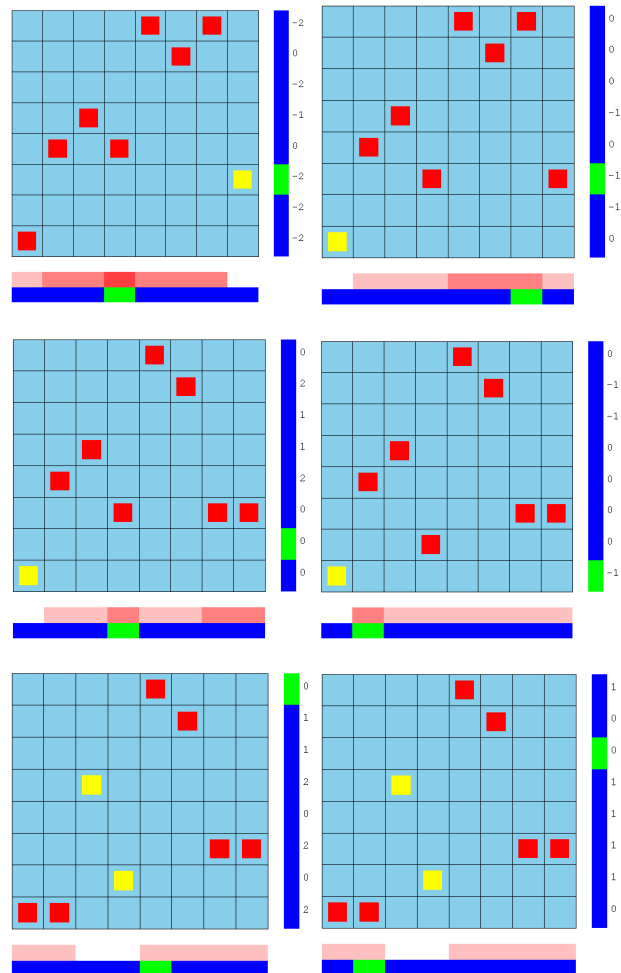
Figure 9: The First Six Steps of the Comet Program

Figure 9 illustrates the computation model and depicts the first six iterations of the algorithm. Figure 10 also shows a plot of the number of violations over time. Each box represents the board and is adorned at the bottom with a color-coded bar showing which queen are currently violating constraints. The color intensity indicates the violation level. Darker tabs correspond to more violations. Below the color-coded bar, a tab shows which queen is picked for the next

Figure 10: The Number of Violations over Time

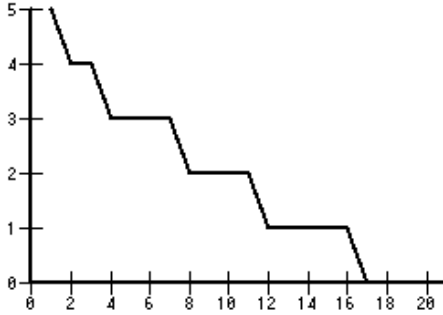| Size	State(s)	Solve(s)	Total(s)	Iter
8	0.01	0.02	0.03	249
16	0.01	0.00	0.01	21
32	0.02	0.01	0.03	56
64	0.04	0.02	0.06	80
128	0.10	0.08	0.18	238
256	0.17	0.12	0.29	206
512	0.35	0.37	0.72	304
1024	0.71	1.64	2.35	628
2048	1.43	6.07	7.50	1100
4096	2.95	22.56	25.51	2092
8192	6.16	85.91	92.07	4040
16384	14.69	333.89	348.58	7968
32768	42.10	1320.43	1362.53	15899

Table 1: Experimental Results for N-Queens.

transition and the tab to the right of the board shows the row the selected queen should be assigned to. Additionally, the column of numbers on the right of each board displays the gain that will result from the transition. In essence, the figure displays many of the properties maintained by the COMET program incrementally. In the first step, queen 4 is selected and randomly moved to position 6 reducing the number of violations by 2. (It could have been moved to positions $\{1, 3, 7, 8\}$). In the second step, queen 7 is moved to position 6, reducing the number of violations by one. Note that, in the fifth iteration, the algorithm decides not to move the selected queen as this is, locally, one of the most profitable moves. As a side note, observe that the result of forbidding such moves is not beneficial in general. Indeed, the algorithm is then forced to consider moves which degrade the value of the objective, it oscillates around 1 and 2 violations, and it takes much longer to terminate.

Table 1 reports some experimental results for the program. The program runs in $O(n)$ space and each iteration takes time $O(n)$, since it is necessary to find the value minimizing the number of conflicts. In general, the propagation step takes constant time because the conflict set is small and does not change very much. The table reports the total CPU time in seconds on a 1.1Ghz PC running Linux, as well as the time for stating the constraints and for the search. It also reports the number of iterations. The results clearly indicate that the implementation is extremely competitive with low-level encodings of the algorithm [6] and is quadratic in time experimentally.

In summary, COMET makes it possible to design a very efficient algorithm at the same level of abstraction as typical constraint programs for the task. Moreover, although the underlying algorithms are quite different, the two solutions share the same structure, compositionality, and modularity. Note that constraint programming solutions take quadratic space, which reduces their applicability to large number of queens.

3.3 The Progressive Party Problem

We now describe a second application, the Progressive Party (PP) problem, to illustrate the functionalities of COMET on more realistic applications. The (PP) problem is interesting for several reasons. On the one hand, it is often used as a benchmark for comparing approaches and algorithms in combinatorial optimization, since it is non-trivial. On the other hand, it features a variety of heterogeneous constraints which makes it an interesting exercise in modeling. The COMET program illustrates the use of a constraint system and combinatorial constraints. It also features a tabu search with an intensification component.

3.3.1 The Problem

The progressive party problem can be described as follows (as is traditional, we assume that host boats have been selected). We are given a set of boats (the hosts), a set of crews (the guests), and a number of time periods. Each host has a limited capacity and each crew has a given size. The problem consists of assigning a host to every guest for each time period in order to satisfy three sets of constraints. First, the capacity of the hosts cannot be exceeded by the guest crews. Second, every guest must visit a different host at each time period. Finally, no two guests can meet more than once over the course of the party.

Figure 11 depicts the core of the COMET program. It receives as inputs two ranges `Hosts` and `Guests` that denote the set of hosts and the set of guests respectively. In addition, it receives two arrays of integers `cap` and `crew` where `cap[h]` is the capacity of host `h` and `crew[g]` is the size of guest `g`. Once again, the neighborhood search algorithm for the PP problem features a declarative and a search component.

3.3.2 The Declarative Component

The declarative component primarily declares the incremental variables and the problem constraints. It features a variety of combinatorial constraints and illustrates how to associate weights with constraints, a common technique in neighborhood search (e.g., [43]). The instructions

```
inc{int} r[p in Periods] = boat[g,p];
S.post(new AllDifferent(r),2);
```

specify that a guest `g` never visits a host more than once over the course of the party. Observe that the `post` method which specifies a weight of 2 for the constraint. The instructions

```
inc{int} c[g in Guests] = boat[g,p];
S.post(new WeightedAtmost(c,crew,cap),2);
```

specify the capacity constraints for period `p`. Constraint

```
WeightedAtmost(c,crew,cap)
```

```
LocalSolver m();
UniformDistribution distr(Hosts);
inc{int} boat[Guests,Periods](m,Hosts) := distr.get();
ConstraintSystem S(m);
forall(g in Guests) {
  inc{int} r[p in Periods] = boat[g,p];
  S.post(new AllDifferent(r),2);
}
forall(p in Periods) {
  inc{int} c[g in Guests] = boat[g,p];
  S.post(new WeightedAtmost(c,crew,cap),2);
}
forall(i in Guests, j in Guests :  j > i) {
  inc{int} ri[p in Periods] = boat[i,p];
  inc{int} rj[p in Periods] = boat[j,p];
  S.post(new MeetAtmost(ri,rj,1));
}
inc{int} vd = S.violationDegree();
m.close();

int tabuLength = 2;
int tabu[Guests,Periods,Hosts] = -1;
Solution solution(m); int best = vd;
int it = 0; int stable = 0;
int stableLimit = 2000;
while (vd) {
  int old = vd;
  selectMax(g in Guests,p in Periods)
          (S.violations(boat[g,p]))
    selectMin(h in Hosts: tabu[g,p,h] <= it ||
              S.getAssignDelta(boat[g,p],h) + vd < best)
      (S.getAssignDelta(boat[g,p],h)) {
      tabu[g,p,boat[g,p]] = it + tabuLength;
      boat[g,p] := h;
      if (vd < old && tabuLength > 2)
        tabuLength = tabuLength - 1;
      if (vd >= old && tabuLength < tbl)
        tabuLength = tabuLength + 1;
    }
  if (vd < best) {
    best = vd; stable = 0;
    solution = new Solution(m);
  } else {
    if (stable == stableLimit) {
      solution.restore();
      stable = 0; it = it + tabuLength;
    }
    it = it + 1; stable = stable + 1;
  }
}
```

Figure 11: The Progressive Party Problem in Comet

is a generalized cardinality constraint [50, 1] which holds if

$$\sum_{i \in \text{Guests}} \texttt{crew[i]} * (\texttt{c[i]} = \texttt{j}) \leq \texttt{cap[j]}$$

for all $j \in$ Hosts. Once again, this constraint is differentiable and captures a subproblem arising in many applications. Finally, the instructions

```
inc{int} ri[p in Periods] = boat[i,p];
inc{int} rj[p in Periods] = boat[j,p];
S.post(new MeetAtmost(ri,rj,1));
```

specify that no two guest crews meet more than once. The constraint meetAtmost(a,b,l) is also a cardinality constraint which holds if

$$\#\{i \in \texttt{R} \mid \texttt{a}[i] = \texttt{b}[i]\} \leq l$$

where R is the range of arrays a and b.

Hosts/Periods	6	7	8	9	10
1-12,16	0.98	1.64	5.13	90.19	
1-13	0.61	0.90	1.17	4.41	21.00
1,3-13,19	0.90	1.53	5.28	253.92	
3-13,25,26	1.21	1.81	7.02	82.66	
1-11,19,21	4.50	24.35			
1-9,16-19	6.20	161.16			

Table 2: Experimental Results for the PP Problem.

3.3.3 The Search Component

The search component is a tabu-search procedure minimizing the violation degree of the constraint system. It first selects a guest crew g and a time period t such that variable boat[g,t] appears in the most conflicts. It then selects a non-tabu host for the pair (g,t) that minimizes the violation degree of the system. The host selection features an aspiration criteria

```
S.getAssignDelta(boat[g,p],h) + vd < best
```

which overrides the tabu status when the assignment improves upon the best "solution" found so far.

The search also includes an *intensification* process. The key idea behind the intensification is to restart from the best found solution when the number of violations has not decreased for a long time. The search uses the solution concept of COMET for implementing the intensification in a simple manner. The instruction

```
Solution solution(m);
```

declares a solution which stores the current value of all decision variables, i.e., incremental variables whose values are not specified by an invariant. Each time a better solution is found, the instruction

```
solution = new Solution(m);
```

makes sure that solution now maintains the best solution found so far. After a number of iterations without improvements, the instruction solution.restore() restores the best available solution and resumes the search from this state. Once again, observe the simplicity of the search component which can be described very concisely.

3.3.4 Experimental Results

Before presenting the results, it is useful to emphasize that the program provides a very high-level and natural modeling of the application. Yet, as the results show, the algorithm compares extremely well with low-level codes developed for this application. The experiments use the same host configurations as in [55] to evaluate the algorithm. For each of these configurations, we consider problems with 6 periods (as in [55]) but also with 7, 8, 9, and 10 time periods. Table 2 reports median CPU Time in seconds for 10 runs of the algorithm. As can be seen, the algorithms easily outperforms the approach in [55].

3.4 Jobshop Scheduling

Many complex applications in areas such as scheduling and routing use complex neighborhoods consisting of several heterogeneous moves. For instance, the elegant tabu-search of Dell'Amico and Trubian [28] consists of the union of two subneighborhoods, each of which consisting of several types

of moves. Similarly, many advanced vehicle routing algorithms [25, 9, 2] use a variety of heterogeneous moves.

The difficulty in expressing these algorithms come from the temporal disconnection between the move selection and execution. In general, a tabu-search or a greedy local search algorithm first scans the neighborhood to determine the best move, before executing the selected move. However, in these complex applications, the exploration cannot be expressed using a (multidimensional) selector, since the moves are heterogeneous and obtained by iterating over different sets. As a consequence, an implementation would typically create classes to store the information necessary to characterize the different types of moves. Each of these classes would inherit from a common abstract class (or would implement the same interface). During the scanning phase, the algorithm would create instances of these classes to represent selected moves and store them in a selector whenever appropriate. During the execution phase, the algorithm would extracts the selected move and apply its `execute` operation. The drawbacks of this approach are twofold. On the one hand, it requires the definition of a several classes to represent the moves. On the other hand, it fragments the code, separating the *evaluation* of a move from its *execution* in the program source. As a result, the program is less readable and more verbose.

COMET supports a `neighbor` construct, which relies heavily on closures and eliminates these drawbacks. It makes it possible to specify the move evaluation and execution in one place and avoids unnecessary class definitions. More important, it significantly enhances compositionality and reuse, since the various subneighborhoods do not have to agree on a common interface or abstract class. The `neighbor` construct are of the form

```
neighbor(δ,N) M
```

where `M` is a move, δ is its evaluation, and `N` is a neighbor selector, i.e., a container object to store one or several moves and their evaluations. COMET supports a variety of such selectors and users can define their own, since they all implement a common interface. For instance, a typical neighbor selector for tabu-search maintains the best move and its evaluation. The `neighbor` instruction queries selector `N` to find out whether it accepts a move of quality δ, in which case the closure of `M` is submitted to `N`.

The `neighbor` construct significantly simplifies the implementation of the effective tabu-search algorithm of Dell'Amico and Trubian (DT) for jobshop scheduling. We first review the basic ideas behind the DT algorithm and then sketch how the neighborhood exploration is expressed in COMET. Algorithm DT uses neighborhood $NC = RNA \cup NB$, where RNA is a neighborhood swapping vertices on a critical path (critical vertices) and NB is a neighborhood where a critical vertex is moved toward the beginning or the end of its critical block. More precisely, RNA considers sequences of the form $\langle p, v, s \rangle$, where v is a critical vertex and p, v, s represent successive tasks on the same machine, and explores all permutations of these three vertices. Neighborhood NB considers a maximal sequence $\langle v_1, \ldots, v_i, \ldots, v_n \rangle$ of critical vertices on the same machine. For each such subsequence and each vertex v_i, it explores the schedule obtained by placing v_i at the beginning or at the end of the block, i.e.,

$$\langle v_i, v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \rangle \vee \langle v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n, v_i \rangle$$

Since these schedules are not necessarily feasible, NB actually considers the leftmost and rightmost feasible positions for v_i (instead of the first and last position). NB is connected which is an important theoretical property of neighborhoods.

We now show excerpts of the neighborhood implementation in COMET. The top-level methods are as follows:

```
void executeMove() {
  MinNeighborSelector N();
  exploreN(N);
  if (N.hasMove())
    call(N.getMove());
}
void exploreN(NeighborSelector N)
{
  exploreRNA(N);
  exploreNB(N);
}
```

Method `executeMove` creates a selector, explores the neighborhood, and executes the best move (if any). Method `exploreN` explores the neighborhood and illustrates the compositionality of the approach: It is easy to add new neighborhoods without modifying existing code, since the subneighborhoods do not have to agree on a common interface or abstract class. The implementation of `exploreRNA` and `exploreNB` is of course where the `neighbor` construct is used.

```
1.   void exploreNB(NeighborSelector N) {
2.    forall(v in _jobshop.getCriticalVertices()) {
3.       int lm = _jobshop.leftMostFeasible(v);
4.       if (lm > 0) {
5.          int delta = _jobshop.moveBackwardDelta(v,lm);
6.          if (acceptNBLeft(delta,v))
7.             neighbor(delta,N)
8.                _jobshop.moveBackward(v,lm);
9.       }
10.      int rm = _jobshop.rightMostFeasible(v);
11.      if (rm > 0) {
12.         int delta = _jobshop.moveForwardDelta(v,rm);
13.         if (acceptNBRight(delta,v))
14.            neighbor(delta,N)
15.               _jobshop.moveForward(v,rm);
16.      }
17.   }
18.}
```

Figure 12: Neighborhood *NB* in COMET.

Figure 12 gives the implementation of `exploreNB`: method `exploreRNA` is similar in spirit, but somewhat more complex, since it involves 5 different moves, as well as additional conditions to ensure feasibility. Method `exploreNB` uses the instance variable `_jobshop`, which is a differentiable object representing the disjunctive graph, a fundamental concept in jobshop scheduling [39]. This differential object maintains the release and tail dates of all vertices, as well as the critical paths, under various operations on the disjunctive graph. The `exploreNB` method iterates over all critical vertices. For each of them it finds the leftmost feasible insertion point in its critical block (line 3). If such a feasible insertion point exists, it evaluates the move (line 5) and then tests if the move is acceptable (line 6). In the DT algorithm, this involves testing the tabu status, a cycling condition, and the aspiration criterion. If the move is acceptable, the `neighbor` instruction is executed. The move itself consists of moving vertex `v` by `lm` positions backwards. Note that, although

	ABZ5	ABZ6	ABZ7	ABZ8	ABZ9	MT10	MT20	ORB1	ORB2	ORB3	ORB4	ORB5
DT	139.5	86.8	320.1	336.1	320.8	155.8	160.1	157.6	136.4	157.3	156.8	140.1
DT*	6.2	3.8	14.2	15.1	14.2	6.9	7.1	7.0	6.0	7.0	6.9	6.2
KS	7.8	8.2	20.7	23.1	20.3	8.7	16.4	9.2	7.8	9.3	8.5	8.1
KS*	4.6	4.8	12.2	13.6	11.9	5.1	9.6	5.4	4.6	5.5	5.0	4.8
CO	5.9	5.7	11.7	9.9	9.0	6.7	9.8	5.6	4.8	5.6	6.3	6.5

Table 3: Computational Results on the Tabu-Search Algorithm (DT)

the move is specified in the `neighbor` instruction, it is not executed. Only the best move is executed and this takes place in method `executeMove` once the entire neighborhood has been explored. The remaining of method `exploreNB` handles the symmetric forward move.

The neighborhood exploration is particularly elegant (in our opinion). Although a move evaluation and its execution take place at different execution times, the `neighbor` construct makes it possible to specify them together, significantly enhancing clarity and programming ease. The move evaluation and execution are textually adjacent and the logic underlying the neighborhood is not made obscure by introducing intermediary classes and methods. Compositionality is another fundamental advantage of the code organization. As mentioned earlier, new moves can be added easily, without affecting existing code. Equally or more important perhaps, the approach separates the neighborhood definition (method `exploreN`) from its use (method `executeMove` in the DT algorithm). This makes it possible to use the neighborhood exploration in many different ways without any modification to its code. For instance, a semi-greedy strategy, which selects one of the k-best moves, only requires to use a semi-greedy selector. Similarly, method `exploreN` can be used to collect all neighbors which is useful in intensification strategies based on elite solutions [36].

Table 3 presents some preliminary experimental results on jobshop scheduling. It compares various implementations of the tabu-search algorithm DT (the goal, of course, is not to compare various scheduling algorithms). In particular, it compares the original results [28], a C++ implementation [40], and the COMET implementation. Table 3 presents the results corresponding to Table 3 in [28]. Since DT is actually faster on the LA benchmarks (Table 4 in [28]), these results are representative. In the table, DT is the original implementation on a 33mhz PC, DT* is the scaled times on a 745mhz PC, KS is the C++ implementation on a 440 MHz Sun Ultra, KS* are the scaled times on a 745mhz PC, and CO are the COMET times on a 745mhz PC. Scaling was based on the clock frequency, which is favorable to slower machines (especially for the Sun). The times corresponds to the average over multiple runs (5 for DT, 20 for KS, and 50 for CO). Results for COMET are for the JIT compiler but include garbage collection. The results clearly indicate that COMET can be implemented to be competitive with specialized programs. Note also that the C++ implementation is more than 4,000 lines long, while the COMET program has about 400 lines.

4. COMET IN CONTEXT

It is worth summarizing the results presented in this paper. *The main message is that, although they support fundamentally different types of algorithms, constraint programming and* COMET *share a common architecture which pro-*

motes modularity, compositionality, reuse, and separation of concerns. The architecture combines declarative and search components which express the problem constraints at a high level of abstraction and allow for concise descriptions of exploration algorithms. As a result, programs in constraint programming languages and COMET often exhibit a similar organization and close modeling components. The search components are quite distinct in general, because of the nature of the underlying algorithms.

Issue	CP	NS
Variables	logical/domain	incremental
Constraints	numeric	numeric
	structural	structural
Search	tree search	graph exploration
	nondeterministic	randomized
	strategies	meta-heuristics
Architecture	layered	layered
Constraints	pruning	differentiability
Search	choice points	closures
	backtracking	inverse functions

Table 4: Contrasting CP and NS

There are of course fundamental technical difference between constraint programming and COMET, some of which are captured in Table 4. The top part of the table discusses conceptual differences, while the bottom part addresses operational concerns.

At the conceptual level, *the key distinction lies in the nature of the variables.* Constraint programming languages for combinatorial optimization are based on logical variables. The values of these variables are unknown initially[1] and it is the purpose of the computation to find these values that satisfy all constraints. COMET, in contrast, relies on incremental variables, i.e., variables which maintain both an old and a new state to facilitate incremental computation. Once these respective variables are in place, the constraints are essentially similar. They capture properties of the solutions, they may be numeric or structural, and they may be composed naturally. As mentioned earlier, search is rather different in both frameworks. Constraint programming heavily relies on a tree-search exploration model, and uses nondeterministic constructs to specify the search tree and search strategies to describe how to explore it. COMET supports graph exploration procedures which rely on randomization, checkpointing, and meta-heuristics. Interestingly, COMET supports the concept of invariants. Invariants provide an intermediate layer between variables and constraints, simplifying the implementation of constraints and other differentiable objects. Such intermediate layers will become in-

---

[1] They generally take their values in a range and, for this reason, are often called *domain variables.*

creasingly important as constraint programming matures. In fact, invariants could already be useful for a variety of purposes in constraint programming, although this has not been explored to date.

At the operational level, the main commonality is the overall architecture which relies on data-driven computations. But the differences are more stricking. Constraints in constraint programming embody pruning algorithms, while they encapsulate incremental algorithms in COMET. Search in constraint programming relies on choice points, back-tracking, and trailing,[2] while COMET uses closures and inverse functions derived from incremental algorithms.

In summary, recent research on COMET seems to indicate that constraint programming and neighborhood search can be supported by high-level languages and libraries with similar abstraction levels, compositionality, and programming style. Obviously, the actual abstractions and their operational semantics are quite distinct, which is to be expected given the nature of the underlying paradigms. But, over the course of the research, novel concepts, as well as novel uses of old ones, have emerged to show remarkable similarities in the way these two paradigms can be supported.

It is very satisfying to look at COMET now and to remember that it started thanks to the gentle encouragements of Paris Kanellakis in the mid 1990s.

5. ACKNOWLEDGMENT

6. REFERENCES

[1] N. Beldiceanu and M. Carlsson. Revisiting the cardinality Operator. In *ICLP-01*, Cyprus, 2001.

[2] R. Bent and P. Van Hentenryck. A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows. *Transportation Science*, 2001. (To Appear).

[3] F. Berthier. Using CHIP to support decision making. In *Actes du Séminaire 1988 - Programmation en Logique*, (Trégastel, France), May 1988.

[4] A. Bockmayr and A. Courtois. Using Hybrid Concurrent Constraint Programming to Model Dynamic Biological Systems. In *ICLP-2002*, pages 85–99, Copenhagen, Denmark, 2002.

[5] A. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transaction on Programming Languages and Systems*, 3(4):353–387, 1981.

[6] C. Codognet and D. Diaz. Yet Another Local Search Method for Constraint Solving. In *AAAI Fall Symposium on Using Uncertainty within Computation*, Cape Cod, MA., 2001.

[7] A. Colmerauer. PROLOG II: Manuel de Reférence et Modèle Théorique. Technical report, GIA - Faculté de Sciences de Luminy, March 1982.

[8] A. Colmerauer. An Introduction to Prolog III. *Commun. ACM*, 28(4):412–418, 1990.

[9] B. De Backer, V. Furnon, P. Shaw, P. Kilby, and P. Prosser. Solving Vehicle Routing Problems Using Constraint Programming and Metaheuristics. *Journal of Heuristics*, 6:501–523, 2000.

[10] L. Di Gaspero and A. Schaerf. *Optimization Software Class Libraries*, chapter Writing Local Search Algorithms Using EasyLocal++. Kluwer, 2002.

[11] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.

[12] R. Fourer, D. Gay, and B. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA, 1993.

[13] P. Galinier and J.-K. Hao. A General Approach for Constraint Solving by Local Search. In *CP-AI-OR'00*, Paderborn, Germany, March 2000.

[14] A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *Proceedings of the First International Conference on Computational Logic*, London, UK, July 2000.

[15] T. Graf, P. Van Hentenryck, C. Pradelles, and L. Zimmer. Simulation of hybrid circuits in constraint logic programming. In *International Joint Conference on Artificial Intelligence*, Detroit, Michigan, August 1989.

[16] V. Gupta, R. Jagadeesan, and V. Saraswat. Hybrid cc, Hybrid Automata, and Program Verification. In *Hybrid Systems*, pages 52–63, 1995.

[17] W. Harvey and M. Ginsberg. Limited Discrepancy Search. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, Montreal, Canada, August 1995.

[18] N. Heintze, S. Michaylov, and P. Stuckey. CLP($\Re$) and some electrical engineering problems. In *the 4th International Conference on Logic Programming*, (Melbourne, Australia), May 1987. The MIT Press.

[19] T. Huynh and C. Lassez. A CLP($\Re$) option trading analysis system. In *the 5th International Conference on Logic Programming*, (Seattle, WA), August 1988. The MIT Press.

[20] Ilog Solver 4.4. Reference Manual. Ilog SA, Gentilly, France, 1998.

[21] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *POPL-87*, (Munich, Germany), January 1987.

[22] P. Kanellakis, G. Hillebrand, and H. Mairson. An Analysis of Core-ML: Expressive Power and Type Inference. In *International Colloquium on Automata, Languages and Programming*, pages 83–105, Jerusalem, Israel, July 1994.

[23] P. Kanellakis, G. Kuper, and P. Revesz. Constraint Query Languages. In *PODS-90*, Nashville, Te, 1990.

[24] P. Kanellakis and C. Papadimitriou. Local Search for the Asymmetric Traveling Salesman Problem. *Operations Research*, 27(3):533–549, 1980.

[25] Kindervater, G. and Savelsbergh, M.W. Vehicle routing: Handling edge exchanges. In E. Aarts and

---

[2]Some implementations also use copying [42].

J. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 482–520. Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons Ltd, England, 1997.

[26] W. Kohn, A. Nerode, and V. Subrhamanian. Constraint Logic Programming: Hybrid Control and Logic as Linear Programming. In V. Saraswat and P. V. Hentenryck, editors, *Principles and Practice of Constraint Programming*, pages 85–100. The MIT Press, Cambridge, Massachussetts, 1995.

[27] F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)*, Pisa, Italy, October 1998.

[28] D. M. and T. M. Applying Tabu Search to the Job-Shop Scheduling Problem. *Annals of Operations Research*, 41:231–252, 1993.

[29] A. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[30] L. Michel and P. Van Hentenryck. Localizer: A Modeling Language for Local Search. *Informs Journal on Computing*, 11(1):1–14, 1999.

[31] L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5:41–82, 2000.

[32] L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. In *Conference on Object-Oriented Programming Systems, Languages, and Applications.*, pages 101–110, Seattle, WA, USA, November 4-8 2002. ACM.

[33] S. Minton, M. Johnston, and A. Philips. Solving Large-Scale Constraint Satisfaction and Scheduling Problems using a Heuristic Repair Method. In *AAAI-90*, August 1990.

[34] U. Montanari. Networks of Constraints : Fundamental Properties and Applications to Picture Processing. *Information Science*, 7(2):95–132, 1974.

[35] A. Nareyek. *Constraint-Based Agents*. Springer Verlag, 1998.

[36] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Management Science*, 42(6):797–813, 1996.

[37] L. Perron. Search Procedures and Parallelism in Constraint Programming. In *Fifth International Conference on the Principles and Practice of Constraint Programming (CP'99)*, Alexandra, VA, October 1999.

[38] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *AAAI-94, proceedings of the Twelth National Conference on Artificial Intelligence*, pages 362–367, Seattle, Washington, 1994.

[39] B. Roy and B. Sussmann. Les problèmes d'ordonnancement avec contraintes disjonctives. Note DS No. 9 bis, SEMA, Paris, France, 1964.

[40] K. Schmidt. Using Tabu-search to Solve the Job-Shop Scheduling Problem with Sequence Dependent Setup Times. ScM Thesis, Brown University, 2001.

[41] C. Schulte. Programming Constraint Inference Engines. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming*, pages 519–533, Linz, Austria, October 1997.

[42] C. Schulte. Comparing trailing and copying for constraint programming. In D. D. Schreye, editor, *Proceedings of the Sixteenth International Conference on Logic Programming*, pages 275–289, Las Cruces, NM, USA, Nov. 1999. The MIT Press.

[43] B. Selman, H. Kautz, and B. Cohen. Noise Strategies for Improving Local Search. In *AAAI-94*, pages 337–343, 1994.

[44] P. Shaw, B. De Backer, and V. Furnon. Improved local search for CP toolkits. *Annals of Operations Research*, 115:31–50, 2002.

[45] H. Simonis and M. Dincbas. Using an extended prolog for digital circuit design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*, pages 165–188, (Munich, Germany), October 1987. IEEE, New York.

[46] G. Smolka. The Oz Programming Model. In J. van Leeuwen, editor, *Computer Science Today.* LNCS, No. 1000, Springer Verlag, 1995.

[47] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[48] P. Van Hentenryck. Visual Solver: A Modeling Language for Constraint Programming. In *Third International Conference on the Principles and Practice of Constraint Programming (CP'97)*, Lintz, Austria, October 1997. (Invited Talk).

[49] P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, Mass., 1999.

[50] P. Van Hentenryck and Y. Deville. The Cardinality Operator: A New Logical Connective and its Application to Constraint Logic Programming. In *ICLP-91*, June 1991.

[51] P. Van Hentenryck and M. Dincbas. Domains in Logic Programming. In *AAAI-86*, Philadelphia, PA, August 1986.

[52] P. Van Hentenryck and L. Michel. Control Abstractions for Local Search. Technical Report CS-03-06, Brown University, April 2003.

[53] P. Van Hentenryck, L. Perron, and J.-F. Puget. Search and Strategies in OPL. *ACM Transactions on Computational Logic*, 1(2):1–36, October 2000.

[54] S. Voss and D. Woodruff. *Optimization Software Class Libraries.* Kluwer Academic Publishers, 2002.

[55] J. Walser. *Integer Optimization by Local Search.* Springer Verlag, 1998.

[56] D. Waltz. Generating Semantic Descriptions from Drawings of Scenes with Shadows. Technical Report AI271, MIT, MA, November 1972.