Constraints, 10, 137–157, 2005
© 2005 Springer Science + Business Media, Inc. Manufactured in The Netherlands.

Control Abstractions for Local Search

PASCAL VAN HENTENRYCK pvh@cs.brown.edu
Brown University, Box 1910, Providence, RI 02912, USA

LAURENT MICHEL ldm@engr.uconn.edu
University of Connecticut, Storrs, CT 06269-2155, USA

Abstract. COMET is an object-oriented language supporting a constraint-based architecture for local search through declarative and search components. This paper proposes three novel and lightweight control abstractions for the search component, significantly enhancing the compositionality, modularity, and reuse of COMET programs. These abstractions, which includes events and checkpoints, rely on first-class closures as the enabling technology. They are especially useful for expressing, in a modular way, heuristic and meta-heuristics, unions of heterogeneous neighborhoods, and sequential composition of neighborhoods.

Keywords: incremental, constraint, local search, neighborhood, heuristic, meta-heuristic, control, abstraction, search

1. Introduction

Historically, most research on modeling and programming tools for combinatorial optimization has focused on systematic search, which is at the core of branch & bound and constraint satisfaction algorithm. It is only recently that more attention has been devoted to programming tools for local search and its variations (e.g., [6, 12, 15, 24, 28, 29]).

COMET [14] is a novel, object-oriented, programming language specifically designed to simplify the implementation of local search algorithms. Comet supports a constraint-based architecture for local search organized around two main components: a declarative component which models the application in terms of constraints, functions, and invariants, and a search component that specifies the search heuristic and meta-heuristic. Constraints, which are a natural vehicle to express combinatorial optimization problems, are *differentiable objects* in COMET: They maintain a number of properties incrementally and they provide algorithms to evaluate the effect of various operations on these properties. Differentiable objects are typically implemented using invariants, a declarative abstraction for incremental data structures pioneered in [15]. The search component then uses these functionalities to guide the local search using multidimensional, possibly randomized, selectors and other high-level control structures. The architecture enables local search algorithms to be high-level, compositional, and modular. It is possible to add new constraints and to modify or remove existing ones, without having to worry about the global effect of these changes. COMET also separates the modeling and search components, allowing programmers to experiment with different search heuristics and meta-heuristics without affecting the problem modeling.

This separation of concerns gives COMET some flavor of aspect-oriented programming [10] and feature engineering [25], since constraints represent and maintain properties across a wide range of objects. COMET has been applied to many applications and can be implemented to be competitive with tailored algorithms, primarily because of its fast incremental algorithms [14].

This paper focuses on the search component and aims at fostering the compositionality, modularity, and genericity of COMET. It introduces three novel control abstractions whose main benefit is to separate, in the source code, components which are usually presented independently in scientific papers. Indeed, most local search descriptions cover the neighborhood, the search heuristic, and the meta-heuristic separately. Yet typical implementations of these algorithms exhibit complex interleavings of these independent aspects and/or require many intermediary classes and/or interfaces. The resulting code is opaque, less extensible, and less reusable. The new control abstractions address these limitations and reduce the distance between high-level descriptions and their implementations.

The first abstraction, *events*, enables programmers to isolate the search heuristic from the meta-heuristic, as well as the algorithm animation from the modeling and search components. The second abstraction, *neighbors*, aims at expressing naturally unions of heterogeneous neighborhoods, which often arise in complex routing and scheduling applications. It allows programmers to separate the neighborhood definition from its exploration, while keeping move evaluation and execution textually close. The third abstraction, *checkpoints*, simplifies the sequential composition of neighborhoods, which is often present in large-scale neighborhood search.

These three control abstractions, not only share the same conceptual motivation, but are also based on a common enabling technology: *first-class closures*. Closures make it possible to separate the definition of a dynamic behavior from its use, providing a simple and uniform implementation technology for the three control abstractions. Once closures are available, the control abstractions really become lightweight extensions, which is part of their appeal.

The rest of this paper is organized as follows. Section 2 briefly reviews the local search architecture and its implementation in COMET. Section 3 gives a brief overview of closures. Sections 4, 5, and 6 present the new control abstractions and sketches their implementation. Section 7 presents some experimental results showing the viability of the approach. Section 8 concludes the paper.

2. The Constraint-Based Architecture for Local Search

This section is a brief overview of the constraint-based architecture for local search and its implementation in COMET. See [14] for more detail. The architecture consists of a declarative and a search component organized in three layers. The kernel of the architecture is the concept of *invariants* over algebraic and set expressions [15]. Invariants are expressed in terms of incremental variables and specify a relation which must be maintained under modifications to its variables. 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) and can be queried to evaluate the effect of local moves on these properties*. They are fundamental because many local 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. *Differentiable objects also 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 local search algorithms. As such, they bring into local 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 also advocated in [3, 7, 18, 29].

These first two layers, invariants and differentiable objects, constitute the declarative or modeling component of the architecture. The third layer of the architecture is the search component which aims at simplifying the implementation of heuristics and meta-heuristics, another critical aspect of local search algorithms. It does not prescribe any specific heuristic or meta-heuristic. Rather, it features high-level constructs and abstractions to simplify the neighborhood exploration and the implementation of meta-heuristics. These includes several multidimensional selectors, abstractions to manipulate solutions, and advanced simulation techniques.

Observe also that the declarative component can be further extended to support vertical extensions that simplify significant classes of implementation. Such a vertical extension for scheduling, which was introduced in [27], provides significant abstractions for both the modeling and the search components.

Figure 1 illustrates the architecture, and its COMET implementation, on the queens problem. The COMET algorithm is based on the max/min-conflict heuristic inspired by [17]. The algorithm 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. Since a queen must be placed on every column, the algorithm uses an array `queen` of variables and `queen[i]` denotes the row of the queen placed on column `i`. Lines 1–4 declare a range, a local solver, a uniform distribution, and an array of incremental variables for representing the row of each queen. *The modeling component* is given in Lines 5–9. Line 5 declares a constraint system and lines 6–8 add the three traditional `alldifferent` constraints, showing how COMET supports "global" combinatorial constraints for local search. *The search component* is given in lines 10–16. It iterates lines 12–15 until the constraint system is true, i.e., no constraint is violated. Line 12 selects a most violated queen, while line 13 selects a new value `v` for the selected queen. The value is selected to minimize the number of violations of the selected queen. To implement this min-conflict

```
1.  range Size = 1..1024;
2.  LocalSolver ls();
3.  UniformDistribution distr(Size);
4.  var{int} queen[i in Size](ls,Size) := distr.get();

5.  ConstraintSystem S(ls);
6.  S.post(alldifferent(queen));
7.  S.post(alldifferent(all(i in Size) queen[i] - i));
8.  S.post(alldifferent(all(i in Size) queen[i] + i));
9.  ls.close();

10. Counter it(ls);
11. while (!S.isTrue()) {
12.   selectMax(q in Size)(S.violations(queen[q]))
13.     selectMin(v in Size)(S.getAssignDelta(queen[q],v))
14.       queen[q] := v;
15.   it++;
16. }
```

Figure 1. The queens problem in COMET.

heuristic, COMET queries the constraint system, a differential object, to find out the effect of assigning queen `q` to each row. Line 14 simply executes the move, automatically updating all invariants and constraints. The operator := assigns values to incremental variables (unlike operator = which is the traditional assignment operator). The use of the counter `it` will become clear later in the paper.

Observe that the search and declarative components are clearly separated in the program. It is thus easy to modify one of them (e.g., adding a constraint and/or changing the search heuristic) without affecting the other. Although the two components are physically separated in the program code, they closely collaborate during execution. The declarative component is used to guide the search, while the assignment `queen[q] := v` starts a propagation phase which updates all invariants and constraints. This compositionality and clear separation of concerns are some of the appealing features of the architecture. *This is precisely such properties which this paper tries to foster further.* Note also 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 them, which is the role of the incremental algorithms in the COMET runtime system.

3. Closures in COMET

Closures are the common enabling technology behind all three control abstractions introduced in this paper. A closure is a piece of code together with its environment. Closures are ubiquitous in functional programming languages, where they are first-class citizens. They are rarely supported in object-oriented languages however. To illustrate the use of closures in COMET, consider the following class

```
class DemoClosure {
  DemoClosure() {}
  Closure print(int i) {
    return new closure
      {cout << i << endl;}
  }
}
```

```
DemoClosure demo();
Closure c1 = demo.print(9);
Closure c2 = demo.print(5);
call(c2);
call(c1);
call(c2);
```

Method `print` receives an integer `i` and returns a closure which, when executed, prints `i` on the standard output. The following snippet shows how to use closures in COMET: the snippet displays 5, 9, and 5 on the standard output. Observe that closures are first-class citizens: They can be stored in data structures, passed as parameters, and returned as results. The two closures created in the example above share the same code (i.e., `cout << i << endl`), but their environments differ. Both contain only one entry (variable `i`), but they associate the value 9 (closure `c1`) and the value 5 (closure `c2`) to this entry. When a closure is created, its environment is saved and, when a closure is executed, the environment is restored before, and popped after, execution of its code. Closures can be rather complex and have environments containing many parameters and local variables, as will become clear later on. It is also worth mentioning that closures can be viewed as first-order functions with no parameters, which is exactly what is needed for the abstractions described later in the paper.

4. Events for Modularity, Compositionality, and Reuse

One of the fundamental benefits of COMET is its ability to separate problem modeling from search. This separation of concerns is made possible by incremental variables, invariants, and differential objects. However, practical applications typically involve other components which would also benefit from such modularity. One such component is algorithm animation, which is valuable early in the development process to visualize the local search behavior. Another component is the meta-heuristic which is often orthogonal and independent from the search heuristic. This section introduces the concept of *publish/subscribe events* in COMET, which make this separation of concerns possible. Informally speaking, classes can publish events, which can be subscribed by event-handlers elsewhere in the code. Methods in the classes can then notify these events, which triggers the event-handler behavior. We first focus on how to use events for animation and meta-heuristic. We then show how to publish and notify events.

4.1. Events for Animation

Consider a graphical animation for the n-queens problem and assume the existence of an `Animation` class handling the graphics and providing a method `updateQueen(int` q, int r) to display the queen on column q and row r. Such an animation is obtained by inserting

```
forall(q in Size)
    whenever queen[q]@changes(int o,int n)
       animation.updateQueen(q,n);
```

just before the search component (between lines 9 and 10). The core is an event-handler that specifies that, whenever the value of queen[q] changes from o to n, the code animation.updateQueen(q,n) must be executed. This event-handler is installed for all queens.

There are a few important points to highlight here. First, the animation code is completely separated from both the modeling and the search components. The glue between the components is the event changes on incremental variables which is *notified* whenever a variable is assigned a new value. The code achieves the same effect as calling animation.updateQueen(q,n) after the assignment of queens, while clearly separating the two aspects and avoiding to clutter the heuristic with animation code. This makes the code more readable and easier to modify and extend. Second, observe that the event-handler behavior animation.updateQueen(q,n) is a closure which depends on the value of q in the environment and is created when the event is subscribed to. Closures make the animation code more natural, avoid the definition of intermediary classes, and feature a textual proximity between the event-handler condition (e.g., the queen is assigned a new value) and its behavior (e.g., update the display of the queen). In traditional object-oriented languages, event conditions and behaviors are separated, which complicates reading and requires new class definitions to store the information necessary to execute the behavior. Finally, observe that events are statically and strongly typed: they enable information to be transmitted from the notifier (e.g., the incremental variable) to the event-handler in a safe fashion with no downcasting.

Events are also compositional. Consider, for instance, adding the functionality of coloring the queens differently according to their number of violations. It is sufficient to add the instructions

```
var{int} violation[q in Size](ls) = S.violations(queen[q]);
forall(q in Size)
    whenever violation[q]@changes(int ov,int nv)
       animation.updateColor(q,nv);
```

This code fragment declares an array of incremental variables maintaining the number of violations of each queen, and updates the color of a queen each time its number of violations is updated. Note that the number of violations of a queen may change even when the queen is not moved. Hence, it is not possible to insert the behavior elsewhere in the program, while remaining incremental, i.e., only considering the queens whose number of violations was modified. This example shows the strengths of events in COMET: they enable elegant animation codes, which would require complex control flows, the creation of intermediary classes, and/or less incrementality in other languages.

4.2. Events for Meta-Heuristics

Events are also beneficial to separate the search heuristic and the meta-heuristic (e.g., tabu-search). They make it possible to divide the statement into modeling, search, and meta-heuristic components. For illustration purposes, consider upgrading the queen algorithm with a tabu-search strategy, which would make a queen tabu for a number of iterations, each time a queen is moved. The tabu-list management can be almost entirely separated from the search heuristic. For instance,

```
1. set{int} tabu();
2. forall(q in Size)
3.  whenever queen[q]@changes(int o,int n) {
4.     tabu.insert(q);
5.     when it@reaches[it+tLen]()
6.        tabu.remove(q);
7.  }
```

shows a simple management of the tabu list, which we now explain in detail. The code declares a set `tabu` to store the tabu queens and features two nested event-handlers. The outermost event-handler is notified each time a queen is moved. It inserts the queen in the tabu set and installs the second event-handler (lines 5–6) whose goal is to remove `q` from the tabu set after `tLen` iterations, where `tLen` is the length of the tabu list. This second handler is interesting in several ways. First, it features a *key-event*, i.e., an event which is parameterized by a specific key which is in between brackets in the code. Here the key is an iteration number and the handler will be notified when the counter `it` will reach or exceed the value `it+tLen`, i.e., the value of the counter when the handler is installed (subscription time) plus the length of the tabu-list. Second, the handler uses the `when` construct, which means that it will be notified only once.

Once this code is in place, the only modification in the search heuristic consists in selecting the queen with the largest number of violations among the non-tabu queens (instead of among all queens). As a consequence, the "glue" between the components (i.e., the counter and the tabu-set) is minimal and the proper behavior is achieved without interleaving the heuristic and the meta heuristic in the source code. Note that, in complex applications, this glue can be anticipated in the first place by assuming that moves are always selected from a restricted set specified by the modeling and/or meta-heuristic components.

4.3. Event Specification and Notification

The examples above focused on the event-handler (the *subscription* part) and showed how the `when` and `whenever` are used to register a behavior. Since they only used primitive objects, no explicit specification and notification of events (the *publish* part) was necessary. Of course, COMET makes it possible to define new events. Each class may publish some events or key-events by declaring them. Its methods are then responsible to notify these events appropriately. To illustrate event specification and notification, consider a possible implementation of the class `Counter` in COMET:

```
  class Counter {
    var{int} _cnt;                        int Counter::++() {
    Event changes(int ov,int nv);           int old = _cnt++;
l   KeyEvent reaches();                     notify changes(old,_cnt);
    Counter(){_cnt=new var{int}(0);}        notify reaches[_cnt]();
    int ++();                               return _cnt;
  }                                       }
```

The class declares an incremental variable `_cnt`, an event `changes` with two parameters, a key-event `reaches` with no parameter, the constructor and the operator. The implementation of the operation (on the right part) notifies the `changes` events, passing the old and new values of the incremental variable. It also notifies all the key-events `reaches`, whose keys are smaller or equal to the value of `_cnt`. These notifications triggers all the event-handlers associated with these events, i.e., it executes the closures which were registered at subscription time by the `when` and `whenever` instructions. In aspect-oriented terms, the `notify` instructions are joint-points and `when` and `whenever` statements are *dynamic* aspects, i.e., aspects associated with instances, not with classes as is typical in aspect-oriented languages.

4.4. Implementation of Events

Conceptually, the implementation of events is close to the OBSERVER design pattern. An event is compiled into virtual machine instructions which explicitly use closures as shown below:

```
when x@changes(int o,int n) ==>  aload x
    cout << n << endl;           newClosure "cout << n << endl;"
                                 subscribeEvent changes,<o,n>
```

The virtual machine is a JVM-like stack machine and `x` and the closure are retrieved from the stack in `subscribeEvent`. At the instance level, each event corresponds to a data structure which collects all the subscribers. Upon notification, the appropriate subscribers are executed, i.e., their parameters are properly initialized and their closures are executed.

5. Union of Heterogeneous Neighborhoods

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 [5] consists of the union of the subneighborhoods, each of which consisting of several types of moves. Similarly, many advanced vehicle routing algorithms [2, 4, 11] use a variety of moves (e.g., swapping visit orders and relocating customers on other routes), each of which may involve a different number of customers and trucks.

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 creates instances of these classes to represent selected moves and stores them in a selector whenever appropriate. During the execution phase, the algorithm extracts the selected move and applies 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.

5.1. The **`neighbor`** *Construct*

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. They key idea is to view a neighbor as a pair $\langle d : int, move : Closure \rangle$ and to have `neighbor` constructs of the form

```
neighbor(d,N) M
```

where `M` is a move, `d` 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 have to implement a common interface. For instance, a typical neighbor selector for tabu-search maintains the best move and its evaluation. The execution of the `neighbor` instruction queries selector `N` to find out whether it accepts a move of quality `d`, in which case the closure of `M` is submitted to `N`.

5.2. Jobshop Scheduling

We now illustrate how the `neighbor` construct significantly simplifies the implementation of the 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.

Before presenting excerpts of the neighborhood code, it is useful to present the model in this scheduling application. The model, depicted in Figure 2, uses the scheduling abstractions presented in [27]. It first declares the modeling objects, i.e., the disjunctive schedule, a fundamental concept in jobshop scheduling [21], the activities, the resources, and the jobs. Lines 6–7 declare the resource constraints, while lines 8–9 specify the precedence constraints. The objective function, the makespan, is declared in Line 10. Both the disjunctive schedule and the makespan are differentiable objects: together they maintain the release and tail dates of all vertices, as well as the critical paths, under various operations on the disjunctive graph.

```
1.  void state() {
2.     sched = new DisjunctiveSchedule(ls);
3.     act = new Activity[i in Activities](sched,duration[i]);
4.     res = new DisjunctiveResource[Machines](sched);
5.     job = new Job[Jobs](sched);

6.     forall(i in Activities)
7.        act[i].requires(res[machine[i]]);
8.     forall(j in Jobs, t in Tasks: t > Tasks.low())
9.        act[jobAct[j,t-1]].precedes(act[jobAct[j,t]],job[j]);
10.    makespan = new Makespan(sched);
11.    sched.close();
12.    ls.close();
13.    source = sched.getSource();
14.    sink = sched.getSink();
15. }
```

Figure 2. Jobshop schedulling: The model.

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

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

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.

Figure 3 gives the implementation of method `exploreNB`. The code of 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` iterates over all critical activities. For each activity (line 3), it finds the leftmost feasible insertion point in its critical block (line 4). If such a feasible insertion point exists (line 5), 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 activity `v` by `lm` positions backwards on its machine (i.e., its disjunctive resource) and update the tabu list. Note that, although 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. It is also worth emphasizing that the move itself refers to activity `act` and to `lm`, both of which will be modified during the subsequent iterations. As a consequence, it should be clear that the code of the move must be a closure.

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. More precisely, this textual proximity has three main benefits.

1. They make the code more readable, since move evaluations and executions are specified in a single location.

```
void exploreNB(NeighborSelector N,set{int} Criticals) {
   forall(v in Criticals) {
      Activity act = act[v];
      int lm = makespan.leftMostCriticalShift(act);
      if (lm > 0) {
         int e = makespan.estimateMoveBackward(act,lm);
         if (tabu.acceptNBL(act,lm,e))
            neighbor(e,N) {
               tabu.updateNBL(act,lm);
               act.moveBackward(lm);
            }
      }
      int rm = makespan.rightMostCriticalShift(act);
      if (rm > 0) {
         int e = makespan.estimateMoveForward(act,rm);
         if (tabu.acceptNBR(act,e))
            neighbor(e,N) {
               tabu.updateNBR(act,rm);
               act.moveForward(rm);
            }
      }
   }
}
```

Figure 3. Exploration of neighborhood *NB* in COMET.

2. They simplify code maintenance, since a single location in the code must be altered when removing, modifying, or adding a move. There is no need to modify code that is spread out in different places.
3. They allow the moves in implicit neighborhoods to resemble explicit moves, which do not separate move evaluations and executions.

It is also important that this proximity between evaluation and execution is similar in essence to events, where the event condition and the event execution are specified together.[1]

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 [19].

5.3. Implementation of Neighbor

The `neighbor` construct is only syntactic sugar once closures are available. Indeed, the syntactic form is rewritten as shown below:

```
forall(v in Size)  ⟹   forall(v in Size)
   neighbor(f,N)          d = f;
      M;                  if (N.accept(d))
                             N.insert(d,new closure { M; });
```

The rewriting uses method `accept` on the selector to determine whether to accept a move. It also ensures that closures are constructed lazily.

6. Sequential Composition of Neighborhoods

This section discusses the use of checkpoint to express the sequential composition concisely. Sequential composition is often fundamental in very large neighborhood search, which explores sequences or trees of (possibly heterogeneous) moves and selects the best encountered neighbor (e.g., [1, 9]). This section illustrates these concepts using variable-depth neighborhood search (VDNS) [9], which was shown very effective on graph-partitioning and traveling salesman problems.

6.1. Variable-Depth Neighborhood Search

VDNS consists of exploring a sequence of moves and moving to the state with best evaluation in the sequence. By exploring sequences which include degrading moves, VDNS may avoid being trapped in poor local optima.

Consider Figure 4 which plots the quality of a sequence of moves. Each node in the graph corresponds to a computation state and two successive nodes are neighbors in the transition graph of the local search. VDNS explores the whole sequence and then returns to the best computation state, i.e., the before-last node.

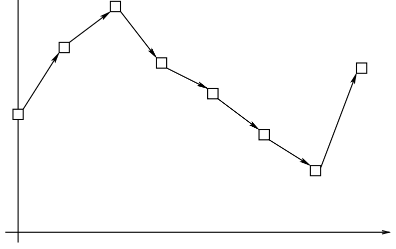

Figure 4. A sequene of moves.

6.2. *Checkpoints*

Checkpoints are a simple conceptual abstraction to express VDNS algorithms. A checkpoint is simply a data structure that implicitly represents the computation state of a local solver, i.e., the state of all incremental variables and data structures of the solver. Whenever a local solver is in checkpointing mode, checkpoints can be saved and, later, restored in order to reset all incremental variables, constraints, and data structures to their earlier states. Checkpoints are first-class citizens in COMET. They also encapsulate incremental algorithms to avoid saving entire computation states.

6.3. *Variable-Depth Neighborhood Search in COMET*

We now illustrate how to express VDNS in COMET for graph partitioning [9], where moves consists of swapping two vertices, one from each set in the partition. The snippet

```
selectBest(ls,nb/2,cost) {
    int mx[i in 0..1] = max(v in A[i] : !mark[v]) gain[v];
    set{int} bL = setof(v in A[0])(!mark[v] && gain[v] == mx[0]);
    set{int} bR = setof(v in A[1])(!mark[v] && gain[v] == mx[1]);
    select(a in bL) {
       selectMax(b in bR)(gain[a] + gain[b] - 2 * adj[a,b]) {
          x[a] :=: x[b];
          mark[a] := true;
          mark[b] := true;
       }
    }
}
```

shows the core of the search procedure in COMET. In the snippet, `ls` is the local solver, `nb` is the number of vertices, `cost` is the cost of the partition, `gain[v]` denotes the gain of changing the set of `v`, and `mark[v]` holds if `v` has already been considered in the VDNS step. The move computes the vertices with maximal gain in each of the partition sets. It then selects a maximal vertex `a` in one of the sets and selects the maximal vertex `b` in the other set such that swapping `a` and `b` minimizes the cost of the partition. The two vertices are then swapped by instruction `a :=: b`. They are also marked to avoid considering them again in this VDNS step. The `selectBest` function is the cornerstone of the VDNS implementation. It receives four arguments: the local solver, the length of the sequence, the function to minimize (an incremental variable), and a closure representing the move.

Figure 5 depicts the implementation of function `selectBest`. It uses the `with checkpoint(ls)` statement to indicate the use of checkpointing inside the enclosed block. It saves the current state in variable `chp` using instruction `Checkpoint`

```
function boolean selectBest(LocalSolver ls,int l,var{int} f,
                            Closure Move)
{
    boolean found = false;
    with checkpoint(ls) {
       Checkpoint chp(ls); int best = f;
       forall(i in 1..l) {
          call(Move);
          if (f < best) {
             found = true;
             best = f;
             chp = new Checkpoint(ls);
          }
       }
       chp.restore();
    }
    return found;
}
```

Figure 5. The implementation of VDNS in COMET.

`chp(ls)`. The `forall` loop explores a sequence of `l` moves, storing the best computation state in variable `chp`. After this exploration, instruction `chp.restore()` restores the best computation state encountered (possibly the initial state). Note that COMET supports the syntactic rewriting from `f(`$a_1,\ldots,a_n$`) S` to `f(`$a_1,\ldots,a_n$`, new closure {S})` when the last argument of function f is a closure. The VDNS implementation has a number of interesting features. First, it is entirely generic and reusable: It can be applied to an arbitrary move and separates search heuristic and the meta-heuristic. Second, checkpoints specify *what* to maintain, i.e., the "best" computation states, but not *how* to save or restore it. The implementation uses incremental algorithms to do so, but this is abstracted from programmers. Finally, observe the role of closures for the genericity of the VDNS implementation.

6.4. Implementation of Checkpoints

We now discuss the checkpoint implementation. The key to an incremental implementation lies in a representation of computation states as sequences of primitive moves from an initial state (i.e., the state when the checkpoint statement is executed). In other words, a state s is a sequence $\langle m_0, \ldots, m_k \rangle$ where m_i is a primitive move. A primitive move in COMET is a function f : *State* $\rightarrow$ *State* from computation states to computation states which is invertible, i.e., there exists a function f^{-1} such that $f(f^{-1}(s)) = s$. For instance, a move `x[i] := j` corresponds to a function $f(s) = s\{x[i]\,/\,j\}$ where $s\{y\,/\,v\}$ represents the state s where y is assigned the value v. The inverse move is of course $f^{-1}(s) = s\{x[i]\,/\,lookup(s^0, x[i])\}$ where s^0 is the computation state before executing the move, and *lookup* reads the value of a variable in a computation state. Consider now how to restore a state s_r from a state s_c where

$$s_c = \langle m_0, \ldots, m_n, m'_{n+1}, \ldots, m'_k \rangle$$
$$s_r = \langle m_0, \ldots, m_n, m''_{n+1}, \ldots, m''_l \rangle.$$

The Comet implementation exploits the common prefix of the two states. It undoes the suffix $\langle m'_{n+1}, \ldots, m'_k \rangle$ by using the inverse moves, and then executes the moves $\langle m''_{n+1}, \ldots, m''_l \rangle$. This implementation has several properties. First, its memory requirements are independent of the size of the computation states. Only moves are memorized and the size of a checkpoint c only depends on the length of the sequence from the initial state to c. Second, the runtime requirements are also minimal, since they either reexecute a subsequence executed before or they execute the inverse of such a subsequence. For VDNS, for instance, restoring the best state does not change the asymptotic complexity: in the worst case, restoring the checkpoint involves as much work as exploring the sequence.

The checkpoint implementation is related to techniques underlying generic search strategies (e.g., [16, 20, 23]). However, it does not use backtracking and/or trailing. Rather, it makes heavy use of inverse moves, which is efficient because the invariant propagation algorithm never updates the same incremental variable twice [15] (which is not the case in constraint satisfaction algorithms in general). Our implementation thus combines low memory requirements with incrementality, which is critical for many local search applications.

7. Experimental Results

This section describes some preliminary experimental results to demonstrate that neighbors and checkpoints can be implemented efficiently.

Table 1. Computational results on the tabu-search algorithm (DT)

	DT	DT*	KS	KS*	CO
ABZ5	139.5	6.2	7.8	4.6	5.9
ABZ6	86.8	3.8	8.2	4.8	5.7
ABZ7	320.1	14.2	20.7	12.2	11.7
ABZ8	336.1	15.1	23.1	13.6	9.9
ABZ9	320.8	14.2	20.3	11.9	9
MT10	155.8	6.9	8.7	5.1	6.7
MT20	160.1	7.1	16.4	9.6	9.8
ORB1	157.6	7.0	9.2	5.4	5.6
ORB2	136.4	6.0	7.8	4.6	4.8
ORB3	157.3	7.0	9.3	5.5	5.6
ORB4	156.8	6.9	8.5	5.0	6.3
ORB5	140.1	6.2	8.1	4.8	6.5

Table 2. Computational results on the graph partitioning

Benchmark	$\min_{SA}$	$\min_{KL}$	μ_{SA}	μ_{KL}	$\#_{SA}$	$\#_{KL}$
Breg500.0	0	0	0	10.6	50	44
Breg500.12	12	12	12	43	50	26
Breg500.16	16	16	16	42.4	50	26
Breg500.20	20	20	20.1	52.5	48	22
Cat.1052	13	31	18.8	45.4	4	4
Cat.352	3	11	5.8	14.9	10	12
G1000.0025	98	116	104	128.9	1	2
G1000.005	450	473	455.5	498.1	2	2
G1000.01	1363	1398	1370.7	1432.9	3	2
G1000.02	3384	3417	3393.7	3468	2	2
G250.01	29	30	31.4	35.2	8	2
G250.02	114	114	115	121	21	2
G250.04	357	359	359.3	367.6	4	2
G250.08	828	828	829.5	841.6	26	4
G500.005	51	58	55.2	65.4	2	2
G500.01	218	229	221.6	243.8	6	2
G500.02	627	635	630.1	653.5	5	2
G500.04	1744	1752	1750.6	1776.8	1	2
Grid.4920	66	60	141.4	60	1	50
Grid.900	30	30	36.3	30	1	50
Grid100.10	10	10	10	10.4	50	46
Grid1000.20	20	20	28.7	20	4	50
Grid500.21	21	21	22.7	21	18	50
Grid5000.50	53	50	126	50	1	50
RCat.134	1	3	2.8	5.1	12	20
RCat.554	3	7	12.1	35.8	6	2
RCat.994	5	5	24.3	52.8	1	2
U1000.05	20	43	36.2	71.2	1	4
U1000.10	50	99	105.4	159.5	1	4
U1000.20	222	222	280.4	278.4	1	4
U1000.40	741	737	1038.8	822.5	1	12
U500.05	7	17	18.6	33.8	1	2
U500.10	26	30	58.1	86.8	2	2
U500.20	178	178	203.2	215.3	2	2
U500.40	442	412	532.6	425.5	1	40
W-grid1000.40	44	40	46.3	40	21	50
W-grid500.42	42	42	44.5	44.9	35	32
W-grid5000.100	108	100	153.9	100	1	50

7.1. Neighbors

This section compares the original results [5], a C++ implementation [22], and the COMET implementation of the tabu search algorithm DT (the goal, of course, is not to compare various scheduling algorithms). The quality of the solutions is similar for all the algorithms (although individual runs may vary due to the randomized selections.)

Table 3. Runtime results on the graph partitioning

Benchmark	$\min_{SA}$ $(s.)$	$\min_{KL}$ (s.)	μ_{SA} $(s.)$	μ_{KL} $(s.)$
Breg500.0	2.79	2.91	2.36	3.3
Breg500.12	3	3.12	2.81	3.96
Breg500.16	3.06	3.16	2.53	4.43
Breg500.20	3.16	3.25	2.61	4.78
Cat.1052	6.87	7.11	14.3	20.19
Cat.352	1.71	1.8	1.64	2.23
G1000.0025	9.01	9.42	10.32	20.63
G1000.005	12.29	12.79	10.08	17.89
G1000.01	18.09	18.7	9.9	17.66
G1000.02	28.72	29.9	13.31	21.74
G250.01	1.67	1.76	0.66	0.95
G250.02	2.09	2.23	0.73	1.11
G250.04	3.06	3.32	1.04	1.46
G250.08	4.94	5.24	1.38	1.89
G500.005	3.9	4.02	2.12	3.82
G500.01	4.83	5.26	2.25	3.22
G500.02	7.44	7.92	2.65	3.84
G500.04	12.89	13.33	4.28	6.6
Grid.4920	44.44	47.64	361.96	433.51
Grid.900	6.73	7	6.06	7.5
Grid100.10	0.37	0.4	0.15	0.18
Grid1000.20	7.35	7.73	7.64	8.92
Grid500.21	3.28	3.43	1.8	2.04
Grid5000.50	44.29	48.58	346.24	441.51
RCat.134	0.47	0.5	0.34	0.41
RCat.554	2.87	3.18	4.3	4.55
RCat.994	5.79	6.82	14.83	15.51
U1000.05	9.08	9.38	8.75	14.63
U1000.10	10.33	11.27	9.04	12.01
U1000.20	11.99	14.17	11.65	13.66
U1000.40	14.98	22.7	16.28	18.46
U500.05	4.22	4.59	1.88	2.79
U500.10	4.89	5.41	2.19	2.79
U500.20	5.98	6.67	3.38	4.2
U500.40	7.05	8.38	5.15	5.86
W-grid1000.40	8.31	8.74	8.67	10.41
W-grid500.42	4.01	4.23	2	2.39
W-grid5000.100	50.34	54.44	334.44	494.6

Table 1 presents the results corresponding to Table 3 in [5]. Since DT is actually faster on the LA benchmarks (Table 4 in [5]), these results are representative. In the table, DT is the original implementation on a 33 mhz PC, DT* is the scaled times on a 745 mhz PC, KS is the C++ implementation on a 440 MHz Sun Ultra, KS* are the scaled times on a 745 mhz PC, and CO are the COMET times on a 745 mhz 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). The Comet code was compiler with the just-in-time compiler and the results for Comet includes 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.

7.2. Checkpoints

Table 2 and 3 report experimental results on graph partitioning problems. In particular, it compares the simulated annealing algorithm of [8] with the variable neighborhood search of [9]. Again, the objective is not to produce the best possible algorithms but rather to indicate that a generic variable-depth search procedure can be implemented efficiently. The two algorithms were evaluated on standard benchmarks from the DIMACS challenge and include various classes and sizes of graphs. Table 2 indicates the best and average solution quality for each algorithm as well as its robustness, i.e., how many runs (out of 50) found the smallest value. Both algorithms produce high-quality solutions in general. The VDNS algorithm is in general comparable to simulated annealing, but it typically produces better solutions on the largest graphs. Table 3 reports the running times (in seconds) and shows that they are usually of the same order of magnitude except on the largest instances. All the benchmarks were executed with the same parameters. Both algorithms use the same parameters on all benchmarks. For simulated annealing, the initial temperature was set at 10 with a cooling ratio of 0.99 (very slow to get high quality solutions). The VDNS algorithm used a maximum of 20 stable iterations as a termination criterion. All the results were obtained on a 2.4 Ghz Pentium 4 processor. These results clearly indicate the practicability of the extensions.

8. Conclusion

This paper presented three novel control abstractions for Comet, which significantly enhance the compositionality, modularity, and reuse of Comet. These abstractions may significantly improve conciseness, extensibility, and clarity of the local search implementations. They all rely on first-class closures as the enabling technology and can be implemented efficiently.

One of the most appealing features of Comet is its small number of fundamental concepts, as well as their generality. First-class closures simplify many applications beyond local search (e.g., [13]) and are ubiquitous in functional programming. Events are related to many constructs in the logic and functional communities (e.g., delay mechanisms and reactive functional programming). Invariants (one-way constraints) and constraints are widely recognized as natural vehicles for many applications. These concepts provide significant support for local search, and may significantly reduce the distance between high-level descriptions of the algorithms and their actual implementations. Yet they are non-intrusive and impose minimal “constraints” on programmers, who retain control of their algorithms and their code organization. An interesting topic for future research is to study how to unify the COMET architecture with the tree-search models proposed in [12, 24], since both approaches have orthogonal strengths.

Acknowledgments

This paper is an extended and revised version of [26]. Comments from the reviewers helped improve the presentation and clarify the design decisions. This work was partially supported by NSF ITR Awards DMI-0121495 and ACI-0121497.

Note

1. Note also that move evaluations can never be separated from move executions. It would make it impossible to retrieve the move leading to that evaluation.

References

1. Balas, E., & Vazacopoulos, A. (1998). Guided local search with shifting bottleneck for job-shop scheduling. *Manage. Sci.* 44(2).
2. Bent, R., & Van Hentenryck, P. (2004). A two-stage hybrid local search for the vehicle routing problem with time windows. *Transp. Sci.* 8: 515–530.
3. Codognet, C., & Diaz, D. (2001). Yet another local search method for constraint solving. In *AAAI Fall Symposium on Using Uncertainty within Computation, Cape Cod, MA*.
4. De Backer, B., et al. (2000). Solving vehicle routing problems using constraint programming and metaheuristics. *Journal of Heuristics*, 6: 501–523.
5. Dell'Amico, M., & Trubian, M. (1993). Applying tabu search to the job-shop scheduling problem. *Ann. Oper. Res.* 41: 231–252.
6. Di Gaspero, L., & Schaerf, A. (2002). Writing local search algorithms using EasyLocal++. In *Optimization Software Class Libraries*, Kluwer Academic Publishers, Boston, MA.
7. Galinier, P., & Hao, J.-K. (2000). A general approach for constraint solving by local search. In *CP-AI-OR'00*, Paderborn, Germany (March).
8. Johnson, D., Aragon, C., McGeoch, L., & Schevon, C. (1989). Optimization by simulated annealing: An experimental evaluation; part I, graph partitioning. *Oper. Res.* 37(6): 865–893.
9. Kernighan, B., & Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *Bell Syst. Tech. J.* 49: 291–307.
10. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., & Irwin, J. Aspect-oriented programming. In *ECOOP '97*.
11. Kindervater, G., & Savelsbergh, M.W. (1997). Vehicle routing: Handling edge exchanges. In *Local Search in Combinatorial Optimization*, Wiley, New York, NY.
12. Laburthe, F., & Caseau. Y. SALSA: A language for search algorithms. In *CP'98*.
13. Manolescu, D. (2002). Workflow enactment with continuation and future objects. In *OOPLSA'02*, Seattle, WA.
14. Michel, L., & Van Hentenryck, P. (2002). A constraint-based architecture for local search. In *OOPLSA'02*, Seattle, WA.
15. Michel, L., & Van Hentenryck, P. (2000). Localizer. *Constraints*, 5: 41–82.

16. Michel, L., & Van Hentenryck, P. (2002). A decomposition-based implementation of search strategies. *ACM Transactions on Computational Logic*.
17. Minton, S., Johnston, M., & Philips, A. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *AAAI-90*.
18. Nareyek, A. (1998). *Constraint-Based Agents*, Springer Verlag, New York, NY.
19. Nowicki, E., & Smutnicki, C. (1996). A fast taboo search algorithm for the job shop problem. *Manage. Sci.* 42(6): 797–813.
20. Perron, L. (1999). Search procedures and parallelism in constraint programming. In *CP'99*, Alexandra, VA.
21. Roy, B., & Sussmann. B. (1964). Les problèmes d'ordonnancement avec contraintes disjonctives. Note DS No. 9 bis, SEMA, Paris, France.
22. Schmidt, K. (2001). Using tabu-search to solve the job-shop scheduling problem with sequence dependent setup times. ScM Thesis, Brown University, Providence, RI.
23. Schulte, C. Comparing trailing and copying for constraint programming. In *ICLP'99*.
24. Shaw, P., De Backer, B., & Furnon, V. (2002). Improved local search for CP toolkits. *Ann. Oper. Res.* 115: 31–50.
25. Turner, C., Fuggetta, A., Lavazza, L., & Wolf, A. (1999). A conceptual basis for feature engineering. *J. Syst. Softw*. 49(1): 3–15.
26. Van Hentenryck, P., & Michel, L. (2003). Control abstractions for local search. In *CP'03*, Cork, Ireland.
27. Van Hentenryck, P., & Michel, Laurent. (2004). Scheduling abstractions for local search. *Proceeding of the First International Conference on the Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CP-AI-OR-04)*, Nice, France, April 2004.
28. Voss, S., & Woodruff, D. (2002). *Optimization Software Class Libraries*, Kluwer Academic Publishers, Boston, MA.
29. Walser, J. (1998). *Integer Optimization by Local Search*, Springer Verlag, New York, NY.