Nondeterministic Control for Hybrid Search

Pascal Van Hentenryck[1] and Laurent Michel[2]

[1] Brown University, Box 1910, Providence, RI 02912
[2] University of Connecticut, Storrs, CT 06269-3155

Abstract. Hybrid algorithms combining local and systematic search often use nondeterminism in fundamentally different ways. They may differ in the strategy to explore the search tree and/or in how computation states are represented. This paper presents nondeterministic control structures to express a variety of hybrid search algorithms concisely and elegantly. These nondeterministic abstractions describe the search tree and are compiled in terms of first-class continuations. They are also parameterized by search controllers that are under user control and specify the state representation and the exploration strategy. The resulting search language is thus high-level, flexible, and directly extensible. The abstractions are illustrated on a jobshop scheduling algorithm that combines tabu search and a limited form of backtracking. Preliminary experimental results indicate that the control structures induce small, often negligible, overheads.

1 Introduction

In the last decade, hybridizations between local and systematic search have received increased attention and contributed many interesting results. Such hybridizations include the use of limited backtracking to intensify local search algorithms around elite solutions [7], variable-depth search procedures that explore trees of moves to select neighbors [1], as well as large neighborhood search where systematic search performs the neighborhood exploration (e.g., [2, 9, 12]).

These hybridizations often lead to fundamentally different search algorithms which may use trailing, copying, incremental checkpointing, or a combination of them in order to restore computation states appropriately. It is therefore a challenge to design flexible, efficient, and elegant search languages to support local and systematic search, as well as their hybridizations.

This paper originated as an attempt to address this challenge in Comet, an object-oriented programming language supporting a constraint-based architecture for local search [5, 14, 15, 16]. It led to the design and implementation of novel nondeterministic abstractions addressing the specificities of hybridizations between local and systematic search, while encompassing the wealth of results in search languages (e.g., [4, 8, 10, 13, 17]).

From a programming standpoint, the nondeterministic control structures of Comet specify the search tree to explore and closely resemble those of OPL. However, these control structures are also parameterized by a search controller

© Springer-Verlag Berlin Heidelberg 2005

that specifies both the search strategy, i.e., how the search tree should be explored, and how search nodes must be stored/restored. Hence, together with other control abstractions of COMET, they provide a rich and flexible search language with several desirable properties.

Perhaps the most significant property is the novel separation between control and state. Indeed, *the control structures only specify nondeterminism. How to explore the resulting search tree and how to store/restore the search nodes are decisions left to the search controller and thus under programmers' control.* This functionality is critical for hybrid search where the state restoration is not necessarily performed using trailing. Instead, state restoration may be based on concepts such as solutions and checkpoints that restore previously saved states with various degrees of incrementality. Note that the separation between control and state also enables different implementation technologies for systematic search (e.g., [11]) to coexist in the same system.

Equally important is the implementation of the nondeterministic control structures which are compiled into continuations in COMET. First-class continuations provide an elegant and efficient abstraction to specify the control flow of nondeterministic abstractions. Moreover, continuations may be implemented to induce no overhead when nondeterminism is not used, which is important for pure local search applications.

Finally, the search language is open and extensible, thanks to continuations and the separation between control and state. Search controllers elegantly implement a variety of systematic search procedures, as well as incomplete search algorithms typically found in hybridizations. Moreover, different state representations, such as trailing, copying, and checkpointing, can be encapsulated inside search controllers, allowing the same nondeterministic abstractions to be used for fundamentally different search procedures.

The rest of this paper introduces the nondeterministic abstractions of COMET. The goal is to convey the rationale underlying their design and implementation, and to illustrate them on a complex application. Section 2 recalls the concept of continuations and illustrates their use in COMET. The nondeterministic abstractions are described in Section 3 and various search controllers are presented in Section 4. The abstractions are illustrated on a hybrid algorithm for jobshop scheduling in Section 5. The last two sections present the experimental results and conclude the paper.

2 Continuations

Continuations provide a flexible control structure to implement several higher-level abstractions such as exceptions, coroutines, and nondeterminism. Informally speaking, a continuation is a snapshot of the runtime data structures that allows the execution to restart from this point at a later stage of the computation. More precisely, a continuation is a pair $\langle I, S\rangle$, where I is an instruction pointer and S is a stack to execute the code starting at I. In COMET, continuations are obtained through instructions of the form

```
continuation c ⟨body⟩
```

```
0.  function int fact(int n) {if (n==0) return 1;else return n*fact(n-1);}
1.  int i = 4;
2.  continuation c { i = 5; }
3.  int r = fact(i);
4.  cout << "fact(" << i << ") = " << r << endl;
5.  if (i == 5) call(c);
```

Fig. 1. Continuations in COMET

that binds c to a continuation $\langle I, S\rangle$, where I is the next instruction in the code and S is the stack when the continuation is captured. It then executes its body and continues in sequence. The resulting continuation can be invoked with the call `call(c)` that restores the stack S and restarts execution from I. Consider the code displayed in Figure 1. The code outputs

```
fact(5) = 120
fact(4) = 24
```

Indeed, the continuation `c` in line 2 consists of an instruction pointer to line 3 and a stack whose entry for `i` stores the value 4. The COMET implementation first calls the factorial function with argument 5 (since `i = 5` is executed when the continuation is taken). Since `i` has value 5, the implementation calls the continuation (line 5), which restarts execution in line 3 with a stack whose entry for `i` has value 4. The COMET implementation thus calls `fact(4)`, displays its result, and terminates (since `i` is 4).

Consider now the code displayed in Figure 2. The code has the same effect but it clearly illustrates the complex control/stack patterns that may be induced by continuations. Indeed, the continuation is taken in line 3, i.e., inside the function `getContinuation` that returns the continuation. The instruction pointer is on line 4 (the `return` instruction) and the stack contains two frames for the global and the function scopes. When the continuation is called on line 9, the stack is restored, the execution restarts in line 4, returns the correct continuation `c`, and proceeds to compute `fact(4)`, displays its results, and terminates. Note that continuations, like closures, are first-class objects that can be stored in data structures, used as arguments, and returned as values.

```
0.  function int fact(int n) {if (n==0) return 1;else return n*fact(n-1);}
1.  int i = 4;
2.  function Continuation getContinuation() {
3.     continuation c { i = 5; }
4.     return c;
5.  }
6.  Continuation c = getContinuation();
7.  int r = fact(i);
8.  cout << "fact(" << i << ") = " << r << endl;
9.  if (i == 5) call(c);
```

Fig. 2. Continuation in COMET Again

3 Nondeterminism

This section describes some of the nondeterministic abstractions of COMET. It assumes initially that nondeterminism is implemented through depth-first search before relaxing this assumption.

The Try Instruction. Figure 3 depicts a nondeterministic program that generates all binary arrays of size 4 and displays their decimal values, i.e., `0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15`. Line 1 simply declares the array and lines 2-7 specify the core of the nondeterministic search. A depth-first search controller i$_S$ created in line 2 and used in all subsequent nondeterministic control structures. Lines 4-5 specify the nondeterministic choices: They iterate over all variables and assign them nondeterministically either to 0 or 1. These lines, as well a$_S$ the output instruction, are encapsulated into an `exploreall` instruction (line 3) in order to produce all solutions. These solutions are obtained by depth-first search, since this is the search controller used in the instruction.

```
0.   include "SearchController";
1.   int x[1..4] = 0;
2.   DFS sc();
3.   exploreall<sc> {
4.     forall(i in 1..4)
5.       try<sc> x[i] = 0; | x[i] = 1;
6.     cout << 8 * x[1] + 4 * x[2] + 2 * x[3] + x[4] << " ";
7.   }
```

Fig. 3. A Simple Nondeterministic Program in Comet

The Exploreall Instruction. The `exploreall` instruction is used to find all solutions to a nondeterministic program. Its implementation simply fails each time it finds a new solution. It is possible to exit an `exploreall` instruction early by using the `exit` method on the search controller.

The Tryall Instruction. Consider now Figure 4 that features a simple backtracking program for the 8-queens problem. The COMET program declares the queens array and the depth-first search controller (lines 1-3) and specifies the search using a `tryall` instruction (lines 4-6). The instructions

```
forall(q in 1..8)
   tryall<dfs>(v in R: !attack(queen,q,v))
      queen[q] = v;
```

iterate over all variables and nondeterministically assign them a value so that the queen in column `q` does not attack the queens in columns `1..q-1`. Observe the iterative style for nondeterminism that is traditionally appreciated by programmers [3]. Observe that the `tryall` instruction performs a nondeterministic choice and then continues the execution normally. Hence it is not only the body of the `tryall` but also the “continuation” of the execution that is nondeterministic.

```
include "SearchController";
int queen[1..8];
DFS dfs();
forall(q in 1..8)
  tryall<dfs>(v in 1..8: !attack(queen,q,v))
    queen[q] = v;
function bool attack(int[] queen,int i, int v) {
  forall(k in 1..i-1)
    if (queen[k]==v || queen[k]+k==v+i || queen[k]-k==v-i)
      return true;
  return false;
}
```

Fig. 4. A Comet Program for the Queens Problem

4 Search Controllers

The nondeterministic instructions only define the search tree to explore. It is the role of the search controller to specify how to explore it, including how to store/restore computation states. This section reviews the interface of search controllers, shows how to compile nondeterminism in terms of the interface and continuations, and reviews a variety of controllers.

The Interface of Search Controllers. Search controllers in Comet are subclasses of `SearchController` which is (partially) described in Figure 5. Several of the methods were informally described earlier. Method `start` is called by `exploreall` to specify what to do when no choice points are left to explore. It receives a continuation that is generally executed in method `exit` that terminates the search. Methods `addChoice` and `fail` constitute the core of the interface and are typically overridden in specific controllers. Method `addChoice` adds a new choice point, while method `fail` restarts execution from an earlier choice point. Observe that choice points are continuations: They primarily specify the control flow, not the computation states.

```
class SearchController {
  Continuation _exit;
  Event closeChoice;
  SearchController() { _exit = null; }
  void start(Continuation e) { _exit = e; }
  void exit() { call(_exit); }
  void addChoice(Continuation c) {}
  void fail() { exit(); }
  ...
}
```

Fig. 5. The Search Controller Interface in Comet (Partial Description)

Compiling Nondeterminism. It is interesting to sketch how nondeterminism is implemented in terms of search controllers and continuations. The `try` instructions are compiled in terms of continuations. A `try` instruction

```
try<sc> ⟨left⟩ | ⟨right⟩
```

is compiled into

```
bool rightBranch = true;
continuation c { sc.addChoice(c); rightBranch = false; ⟨left⟩ }
if (rightBranch) ⟨right⟩
```

The implementation creates a continuation `c`, transfers it to the search controller as a new choice and then executes the left branch of the choice point. On backtracking, i.e., when the continuation `c` is invoked, the right branch is executed since `rightBranch` is true in that context. The compilation of a `tryall` instruction is essentially similar but uses iterators to assign its parameter that is represented as a local variable. An `exploreall` instruction

```
exploreall<sc> ⟨body⟩
```

is compiled into

```
continuation c { sc.start(c); ⟨body⟩; sc.fail(); }
```

The implementation takes a continuation `c` representing what must be executed when no more choice points are left unexplored, i.e., when all the solutions of its body have been explored. It stores the continuation in the search controller, executes the body of the instruction, and fails, which induces the search controller to consider unexplored choices. Note also that method `exit` on the controller invokes the continuation `c` by default.

A Simple Search Controller. Figure 6 depicts the depth-first search controller used so far. The controller maintains a stack of continuations. Method `addChoice` pushes the continuation on the stack, while method `fail` pops and invokes the top continuation. Nothing else is necessary for solving the queens problem. Indeed, the continuation automatically saves the parameters of the `forall` and `tryall` instructions as they are stored on the stack. Moreover, the values of the queens do not need to be restored because of the depth-first strategy.

```
class DFS extends SearchController {
  Stack{Continuation} stack;
  DFS(): SearchController() { stack = new Stack{Continuation};}
  void addChoice(Continuation f) { stack.push(f); }
  void fail() { if (stack.empty()) exit() else call(stack.pop()); }
}
```

Fig. 6. The Depth-First Search Controller

(Re)storing Search Nodes. In more complex applications or with other strategies, the search controllers must save and restore additional data structures. Figure 7 revisits the simple nondeterministic program presented earlier. It now declares a local solver (line 1) and incremental variables (line 2), since this is the underlying technology for the jobshop scheduling application described subsequently. Note the `try` instruction that assigns incremental variables using `:=` instead of `=`.

```
include "LocalSolver";
LocalSolver mgr();
var{int} x[1..3](mgr) := 0;
SDFS sc(mgr);
exploreall<sc> {
  forall(i in 1..3)
    try<sc> x[i] := 0; | x[i] := 1;
  cout << 8 * x[1] + 4 * x[2] + 2 * x[3] + x[4] << endl;
}
```

Fig. 7. The Simple Nondeterministic Program Revisited

The program also declares a depth-first search controller whose implementation is depicted in Figure 8. The controller stores, not only continuations, but also the states of the incremental variables. Line 2 in Figure 8 declares a stack of continuations and a stack of solutions. Solutions in COMET capture a snapshot of the incremental variables, which can then be restored at a later computation stage [5]. When a choice point is created (method `addChoice`), the controller captures a solution (`new Solution(m)`) and pushes it onto the solution stack. On backtracking, the instruction `sol.pop().restore()` restores the solution.

Observe the decoupling between the control and data aspects of choice points. The control flow is abstracted by continuations that are used by the controller to implement the search strategy. The representation of search nodes is under user control and may use abstractions such as solutions, checkpoints, and computation spaces [10]. In other words, the controller describes how to save and restore the nodes independently of the specifications of search tree and the search strategy. As a consequence, the node representation can be changed by replacing or modifying the controller without affecting the rest of the program. For instance, to use checkpoints [14] instead of solutions, it suffices to store checkpoints on the stack using instructions such as `new Checkpoint(m)`. Checkpoint restorations undo and, possibly reexecute, operations on incremental variables as described in [14]. Similar issues arise in CP systems. Trailing-based systems may

```
0.   class SDFS extends SearchController {
1.     LocalSolver m; Stack{Continuation} cont; Stack{Solution} sol;
2.     SDFS(LocalSolver mgr): SearchController() {
3.       m = mgr;
4.       cont = new Stack{Continuation}; sol = new Stack{Solution};
5.     }
6.     void addChoice(Continuation f) {
7.       cont.push(f); sol.push(new Solution(m)); }
8.     void fail() {
9.       if (cont.empty()) exit();
10.      else { sol.pop().restore(); call(cont.pop()); }
11.    }
12.  }
```

Fig. 8. The Depth-First Search Controller with Solutions

```
class SDS extends SearchController {
  LocalSolver m; Queue{Continuation} cont; Queue{Solution} sol;
  SDS(LocalSolver mgr): SearchController() {
    m = mgr;
    cont = new Queue{Continuation}; sol = new Queue{Solution};
  }
  void addChoice(Continuation f) {
    cont.push(f); sol.push(new Solution(m)); }
  void fail() {
    if (cont.empty()) exit();
    else { sol.pop().restore(); call(cont.pop()); }
  }
}
```

Fig. 9. The Discrepancy Search Controller

use CP checkpoints that capture trail pointers and use semantic decomposition for strategies [6], while copy-based systems only save the state of the solver.

A Simple Discrepancy Controller. Figure 9 describes a controller where the stack was replaced by queue, implementing a search strategy where the choices are explored by increasing number of discrepancies. By replacing `SDFS` by `SDS` in line 3 of Figure 7, the program displays `0 8 4 2 1 12 10 9 6 5 3 14 13 11 7 15`. *Observe the simplicity of moving from depth-first search to this new search strategy thanks to the high-level control and state abstractions of* Comet. The control and the states are fully abstracted by continuations and solutions, providing a concise and elegant specification of the search strategy.

An Iterative Discrepancy Controller. Figure 10 depicts a search controller for an iterative implementation of limited discrepancy search. The exploration strategy consists of a sequence of searches, where the i-th search allows at most i discrepancies. The controller maintains a variable `_discr` to count the number of discrepancies that is incremented in method `fail` each time a new choice is explored. Method `fail` only calls a continuation whenever the maximum number of discrepancies is not exceeded (line 22). Otherwise, the controller recursively fails to explore another choice with fewer discrepancies. Note that line 20 pops the continuation and restores the values of the incremental variables, including the number of discrepancies.

The discrepancy phases are initiated in the overridden `start` method. Interestingly, it also uses a `tryall` instruction to explore all the discrepancies up to the maximum depth (line 14). Observe that, like in the queens problem, the nondeterminism operates not only on the body of the `tryall` but also on whatever follows the `start` method (e.g., the body of an `exploreall` instruction). The first two phases of the resulting program display `0 1 2 4 8 0 1 2 3 4 5 6 8 9 10 12`.

A Memento Controller. Figure 11 depicts the search controller to be used in jobshop scheduling. The key idea underlying the controller is to only store the

```
class IDS extends SearchController {
  LocalSolver m;
  Stack{Continuation} cont;
  Stack{Solution} sol;
  var{int} _discr;
  int _maxDiscr;
  int _maxDepth;
  SIDS(LocalSolver mgr,int d) : SearchController() {
    m = mgr; _maxDepth = d;
    cont = new Stack{Continuation}(); sol = new Stack{Solution}();
    _discr = new var{int}(mgr) := 0;
  }
  void start(Continuation e) {
    super.start(e);
    tryall<this>(d in 1.._maxDepth) { _discr := 0; _maxDiscr = d; }
  }
  void addChoice(Continuation f) { cont.push(f); sol.push(f); }
  void fail() {
    if (cont.empty()) exit();
    else {
      Continuation c = cont.pop(); sol.pop().restore();
      _discr := _discr + 1;
      if (_discr > _maxDiscr) fail(); else call(c);
    }
  }
}
```

Fig. 10. The Iterative Discrepancy Search Controller

```
class Memento extends SearchController {
  LocalSolver m;
  Stack{Continuation} cont;
  Stack{Solution} sol;
  int _maxSize;
  Memento(LocalSolver mgr,int maxSize) : SearchController() {
    m = mgr; _maxSize = maxSize;
    cont = new Stack{Continuation}; sol = new Stack{Solution};
  }
  void addChoice(Continuation f) {
    if (cont.getSize() == _maxSize) { cont.drop(); sol.drop(); }
    cont.push(f); sol.push(new Solution(m));
  }
  void fail() {
    if (cont.empty()) exit();
    else { sol.pop().restore(); call(cont.pop()); }
  }
}
```

Fig. 11. The Memento Search Controller

last k choice points. If a new choice point f is available, and there are already k choice points, the controller first drops the earliest stored choice point before pushing f onto the stack. This controller, called `Memento` in the following, is a simple modification to the depth-first controller. Indeed, method `addChoice` now tests whether the maximum number of choices is reached, in which case the earliest choice is dropped from the stack before the push. Once again, observe the simplicity of the controller and the flexibility of the abstractions to implement incomplete search procedures in a natural fashion.

5 An Hybrid Search for Jobshop Scheduling

This section presents an implementation in COMET of the hybrid algorithm of Nowicki and Smutnicki for jobshop scheduling [7]. The algorithm is a tabu-search procedure with a very interesting intensification component based on a limited form of backtracking. Informally speaking, the algorithm maintains the k best solutions found so far. Whenever the tabu search completes, it backtracks to one of these k solutions and explores all its neighbors by restarting a tabu search from each of them. Of course, these new tabu searches may introduce new choice points that will be explored subsequently on backtracking. The rest of this section presents the core of this hybrid search, using the control and scheduling abstractions presented in [14, 16].

Figure 12 presents the core of the algorithm. The search procedure is organized as a series of phases (lines 6-16). Each phase terminates after `curIter` iterations (line 16) or whenever a new best solution is found (line 11). A phase (lines 7-15) consists in exploring the neighborhood and selecting the best move (lines 3, 8-9). Observe the declaration of a `MinNeighborSelector` object (line

```
1.   void JobshopAlgorithm::search() {
2.     memento = new Memento(mgr,mementoSize);
3.     MinNeighborSelector N();
4.     int li = 0;
5.     exploreall<memento> {
6.       do {
7.         int oldBest = bestSoFar;
8.         if (exploreNeighborhood(N)) {
9.           call(N.getMove());
10.          if (oldBest > makespan.value()) {
11.            li = 0; bestSoFar = makespan.value(); curIter = maxIter;
12.            visitAllNeighbors();
13.          }
14.        } else memento.exit();
16.      } while (li++ < curIter);
17.    }
18.  }
```

Fig. 12. Hybrid Search for Jobshop Scheduling

```
1.   void JobshopAlgorithm::visitAllNeighbors() {
2.     AllNeighborsSelector neighborhood();
3.     exploreNeighborhood(neighborhood);
4.     tryall<memento>(i in 1..neighborhood.getSize()-1)
5.       call(neighborhood.getMove(i));
6.   }
```

Fig. 13. Jobshop Scheduling: The Intensification

3) that is passed to the neighborhood exploration (line 8) to return the best move. The selected move, which is a closure, is executed in line 9: It performs the moves and updates the tabu list. If it improves the best solution, the phase is terminated in line 11 (ignore line 12 for the time being). The neighborhood exploration may return false, meaning that the current solution is optimal (i.e., it satisfies a necessary condition for optimality). When this is the case, the search terminates by calling method `exit` on the `memento` declared in line 2.

As mentioned earlier, one of the most innovative aspects of this algorithm is its intensification: Its goal is to explore the search space around elite solutions more extensively. The algorithm maintains a stack of the best k solutions found during the search. When the tabu search completes, the algorithm pops the best solution from the stack and restarts a phase of the tabu search from each of its neighbors. The algorithm terminates when the stack is empty. Observe however that each additional phase may find new best solutions that are themselves pushed onto the stack. The intensification is featured in line 12 of Figure 12 and the implementation is depicted in Figure 13.

The implementation declares another neighbor selector to retrieve all neighbors (line 2). Method `exploreNeighborhood`, called with this selector, collects all neighbors sorted by decreasing quality (line 3). Once the neighbors are available, the nondeterministic instruction `tryall` explores all of them nondeterministically, except the best one that has been explored by the tabu search previously.

This implementation is particularly elegant for several reasons. First, it captures the essence of the intensification concisely and naturally. It only needs four lines of code to express a sophisticated intensification component. Second, the search strategy is completely disconnected from the intensification. If a different search strategy (e.g., depth-first search) is desired, it suffices to replace the memento by a depth-first search controller. Third, the implementation is completely generic: It does not explicitly refer to the neighborhood and changes to the neighborhood (in method `exploreNeighborhood`) do not affect the intensification. Finally, observe that method `exploreNeighborhood` is used both to find the best neighbor (line 8 in Figure 12) and all the neighbors (line 3 in Figure 13) by passing different neighbor selectors.

The original algorithm in [7] also includes another interesting feature: It reduces the length of the phases each time all neighbors of a choice point have been explored. This functionality can be elegantly accommodated in the search procedure by adding the instruction

```
whenever memento@closeChoice() curIter = curIter - 400;
```

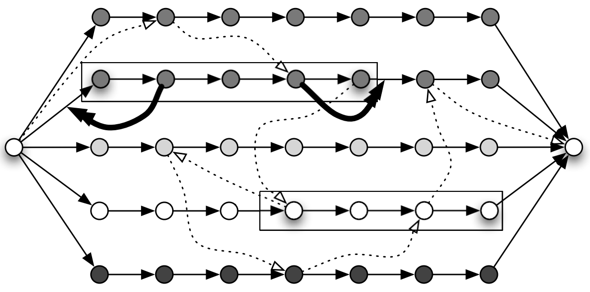

Fig. 14. The Neighborhood of the Jobshop Algorithm

between lines 4 and 5 in Figure 12. This instruction features an event [14] that specifies that, whenever all alternatives of a `tryall` instruction are exhausted (event `closeChoice` of the search controller class), `curIter` must be reduced by 400. Observe the compositionality of the search language and the synergy between the existing and novel control abstractions of Comet.

For completeness, it is useful to discuss the neighborhood briefly. The neighborhood focuses on one critical path from the source to the sink only. Moreover, only the critical arcs at the start or at the end of a critical block are considered for swapping. In other words, the neighborhood identifies the activities at the start (resp. at the end) of a critical block on the selected path and considers the moves that swap such activities with their successors (resp. predecessors). The

```
bool JobshopAlgorithm::exploreNeighborhood(NeighborSelector N) {
  set{Activity} SB();
  set{Activity} EB();
  bool optimal = collectBlockEdges(SB,EB);
  if (!optimal)
    forall(v in SB) {
        Activity s = v.getSucc();
        int eval = makespan.estimateMoveForward(v);
        if (acceptMove(v,s,eval)) {
          found = true;
          neighbor(eval,N) { v.moveForward(); tl.makeTabu(s,v); }
        }
      }
      forall(v in EB) {
        Activity p = v.getPred();
        int eval = makespan.estimateMoveBackward(v);
        if (acceptMove(p,v,eval)) {
          found = true;
          neighbor(eval,N) { v.moveBackward(); tl.makeTabu(v,p); }
        }
      }
  return !optimal;
}
```

Fig. 15. Jobshop Scheduling: The Neighborhood Exploration

resulting neighborhood is not connected, although it is very effective in practice. Moreover, if there is only one critical block on the selected path, then the moves cannot decrease the length of the makespan and the solution can be shown to be optimal. The neighborhood is described visually in Figure 14. In the figure, the machines are depicted horizontally and the job precedences are shown by dashed arcs. The large bold arrows show the two moves associated with the first block. Figure 15 describes parts of the neighborhood exploration which specifies what the neighborhood is, not how to use it. The key for this separation of concerns is the `neighbor` construct [14] which uses closures to represent moves (lines 13-16).

6 Experimental Results

This section presents some preliminary results on the performance of nondeterministic control structures. The first test compares the nondeterministic program in Figure 4 (N in the following) with the "traditional" recursive algorithm R depicted in Figure 16 (the function `noattack` is similar and not shown here). Since no state information is saved in the traditional implementation, this experiment captures the cost of the control structures. It is a worst-case scenario since, in sophisticated applications, the control cost is typically amortized by the state saving and restoration, as well as by propagation and/or the maintenance of incremental data structures. Table 1 depicts the CPU times of the recursive and nondeterministic algorithms for various n. The results indicate that the overhead of the nondeterministic implementation is small and ranges from 7 to 25% which is very reasonable for such a worst case scenario. Observe also the contrast between the iterative style of the nondeterministic program and the recursive style of the traditional program.

```
int n = 8;
range R = 1..n;
int queen[R];
function bool search(int[] queen,int i) {
  if (i > n) return true;
  forall(v in R: !attack(queen,i,v)) {
    queen[i] = v;
    if (search(queen,i+1)) return true;
  }
  return false;
}
search(queen,1);
```

Fig. 16. A Recursive Backtracking Algorithm for the Queens Problem

Table 2 presents some experimental results on jobshop scheduling. It reports the quality and performance of the algorithm on the LA instances. Each line correspond to 50 runs of the algorithm and report statistics on the best, worst,

Table 1. Performance Evaluation of the Nondeterministic Control Instructions

n	16	18	20	22	24	26
R	0.18	0.79	4.78	51.71	14.11	15.52
N	0.20	0.98	5.57	56.80	15.20	16.76
$(N-R)/R$	11.11	24.05	16.53	9.84	7.73	7.99

Table 2. Performance Results on Jobshop Scheduling

Bench	$B(V)$	$m(V)$	$M(V)$	$\mu(V)$	$\sigma(V)$	$m(B)$	$M(B)$	$\mu(B)$	$\sigma(B)$
LA19	842	842 (40)	846	842.4	1.0	0.31	13.45	4.20	3.19
LA20	902	902 (50)	902	902.0	0.0	0.17	2.97	0.95	0.62
LA21	1047	1047 (4)	1061	1052.8	2.7	1.05	19.29	6.56	4.39
LA22	927	930 (6)	939	934.4	2.2	1.53	11.26	5.45	2.72
LA23	1032	1032 (50)	1032	1032.0	0.0	0.34	1.01	0.60	0.14
LA24	935	938 (1)	944	943.3	1.5	0.93	12.39	4.54	2.96
LA25	977	977 (8)	986	980.0	2.6	1.42	17.59	6.14	4.01
LA26	1218	1218 (50)	1218	1218.0	0.0	1.61	6.55	3.24	1.21
LA27	1235	1236 (1)	1269	1251.2	6.9	3.30	23.37	9.66	4.64
LA28	1216	1216 (46)	1225	1216.5	1.9	1.95	23.07	7.46	4.07
LA29	1157	1163 (3)	1190	1174.8	7.1	3.12	33.78	11.77	5.68
LA30	1355	1355 (50)	1355	1355.0	0.0	0.68	2.01	1.40	0.36
LA31	1784	1784 (50)	1784	1784.0	0.0	1.59	5.80	3.26	0.78
LA32	1850	1850 (50)	1850	1850.0	0.0	1.41	6.23	3.66	1.00
LA33	1719	1719 (50)	1719	1719.0	0.0	0.27	3.58	1.19	0.69
LA34	1721	1721 (50)	1721	1721.0	0.0	2.73	7.29	4.91	1.32
LA35	1888	1888 (50)	1888	1888.0	0.0	0.27	4.24	2.03	0.73
LA36	1268	1268 (24)	1291	1272.4	5.7	2.25	20.51	9.53	4.55
LA37	1397	1402 (3)	1428	1412.8	5.6	3.28	32.31	13.35	6.55
LA38	1196	1196 (13)	1208	1200.7	3.1	3.14	29.83	9.39	6.12
LA39	1233	1233 (23)	1251	1237.0	5.2	3.66	28.50	13.49	6.74
LA40	1222	1226 (5)	1234	1231.2	2.8	2.93	24.24	10.36	5.89

average, and standard deviation of the solution quality and CPU times. The number of times the best solution was found is also reported. On this algorithm, the overhead of nondeterminism is not noticeable, since restoring a solution amounts to recomputing the makespan and critical arcs, which is much more costly than creating and restoring continuations.

7 Conclusion

This paper presented nondeterministic control structures for hybrid search procedures which often differ in their underlying node selection strategies and their implementation of search nodes. From a modeling standpoint, the main contribution of the abstraction is to decouple the specification of the search tree, the node selection, and the node representation. In particular, the nondeterministic abstractions separate the specification of the search tree (i.e., the computations to be explored), the control flow (i.e., how the computations are actually explored), and the node representation (i.e., how the search nodes are stored and restored). All these aspects of search procedures remain under programmers' control, combining a high-level iterative style with the flexibility and extensibility necessary to implement a variety of search procedures. From an implementation standpoint, the nondeterministic control structures are compiled into first-order continuations, inducing no overhead when nondeterminism is not used. The expressiveness and practicability of the abstractions was demonstrated by presenting several search controllers, a tabu procedure for job-shop scheduling featuring an intensification based on backtracking search, and unit performance tests to estimate the cost of continuations.

Acknowledgments. This work was partially supported by NSF ITR Awards ACI-0121497.

References

1. A. Balas, E.; Vazacopoulos. Guided Local Search with Shifting Bottleneck for Job Shop Scheduling. *Management Science*, 44(2):262–275, 1998.
2. R. Bent and P. Van Hentenryck. A Two-Stage Hybrid Local Search for the Vehicle Routing Problem with Time Windows. *Transportation Science*, 8(4):515–530, 2004.
3. de Givry, S. and Jeannin, L. Tools: A library for partial and hybrid search methods. In *CP-AI-OR'03*, Montreal, Canada, 2003.
4. F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *CP'98*, Pisa, Italy, October 1998.
5. L. Michel and P. Van Hentenryck. A Constraint-Based Architecture for Local Search. In *OOPSLA'02.*, pages 101–110, Seattle, WA, USA, November 4-8 2002.
6. L. Michel and P. Van Hentenryck. A Decomposition-Based Implementation of Search Strategies. *ACM Transactions on Computational Logic*, 5(2):351-383, 2004.
7. E. Nowicki and C. Smutnicki. A Fast Taboo Search Algorithm for the Job Shop Problem. *Management Science*, 42(6):797–813, 1996.
8. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *CP'99*, pages 346–360, Alexandra, VA, October 1999.
9. L.M. Rousseau, M. Gendreau, and G. Pesant. Using Constraint-Based Operators to Solve the Vehicle Routing Problem with Time Windows. *Journal of Heuristics*, 8:43–58, 2002.
10. C. Schulte. Programming Constraint Inference Engines. In *CP'97*, 519–533, Linz, Austria, October 1997.
11. Christian Schulte. Comparing Trailing and Copying for Constraint Programming. In *ICLP-99*, pages 275–289, Las Cruces, NM, USA, November 1999.
12. P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In *CP'98*, pages 417–431, Pisa, Italy, October 1998.
13. P. Van Hentenryck. *The OPL Optimization Programming Language.* The MIT Press, Cambridge, Mass., 1999.
14. P. Van Hentenryck and L. Michel. Control Abstractions for Local Search. In *CP'03*, pages 65–80, Cork, Ireland, 2003.

15. P. Van Hentenryck and L. Michel. Scheduling Abstractions for Local Search. In *CP-AI-OR'04*, pages 319–334, Nice, France, 2004.
16. P. Van Hentenryck, L. Michel, and L. Liu. Constraint-based Combinators for Local Search. In *CP'04*, Toronto, Canada, 2004.
17. 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.