High-Level Nondeterministic Abstractions in C++

Laurent Michel[1], Andrew See[1], and Pascal Van Hentenryck[2]

[1] University of Connecticut, Storrs, CT 06269-2155
[2] Brown University, Box 1910, Providence, RI 02912

Abstract. This paper presents high-level abstractions for nondeterministic search in C++ which provide the counterpart to advanced features found in recent constraint languages. The abstractions have several benefits: they explicitly highlight the nondeterministic nature of the code, provide a natural iterative style, simplify debugging, and are efficiently implementable using macros and continuations. Their efficiency is demonstrated by comparing their performance with the C++ library Gecode, both for programming search procedures and search engines.

1 Introduction

The ability to specify search procedures has been a fundamental asset of constraint programming languages since their inception (e.g., [1, 3, 13]) and a differentiator compared to earlier tools such as Alice [7] and MIP systems where search was hard-coded in the solver. Indeed, by programming the search, users may define problem-specific branching procedures and heuristics, exploit unconventional search strategies, break symmetries dynamically, and specify termination criteria for the problem at hand. The last two decades have also witnessed significant progress in this area (e.g., [6, 8, 9, 12, 14, 15]): Modern constraint programming languages enable programmers to specify both the search tree and the search strategy, provide high-level nondeterministic abstractions with dynamic filtering and ordering, and support hybrid and heuristic search.

The embedding of constraint programming in mainstream languages such as C++ has also been a fundamental step in its acceptance, especially in industry. With constraint programming libraries, practitioners may use familiar languages and environments, which also simplifies the integration of a constraint programming solution within a larger application. Ilog Solver [10] is the pioneering system in this respect: it showed how the nondeterministic abstractions of constraint logic programming (e.g., goals, disjunction, and conjunction) can be naturally mapped into C++ objects. To specify a search procedure, users thus define C++ objects called goals, and combine them with logical connectives such as `or` and `and`. In recent years, constraint programming libraries have been enhanced to accommodate search strategies [9, 4] (originally proposed in Oz [12]) and high-level nondeterministic abstractions [8] (originally from OPL [14]).

However these libraries, while widely successful, still have two inconveniences as far as specifying search procedures. On the one hand, they impose a recursive style for search procedures, which contrasts with the more familiar iterative constructs of OPL as indicated in [2]. Second, these libraries may obscure

the natural nondeterministic structure of the program and may produce some non-trivial interleaving of C++ code and library functions. This complicates the debugging process which alternates between library and user code.

This paper is an attempt to mirror, in constraint programming libraries, the high-level nondeterministic abstractions of modern constraint programming languages. The paper shows that it is indeed possible and practical to design a search component in C++ that

- promotes an iterative programming style that expresses both sequential composition and nondeterminism naturally;
- simplifies the debugging process, since the C++ stack now reflects the full control flow of the application;
- is as efficient as existing libraries.

The technical idea underlying the paper is to map the nondeterministic abstractions of COMET [15] into C++ using macros and continuations. Obviously, since continuations are not primitive in C++, it is necessary to show how they can be implemented directly in the language itself. The implementation differs significantly from the OPL implementation in which the abstractions are implemented using Ilog Solver facilities.

The rest of the paper is organized as follows. Section 2 presents the nondeterministic abstractions and their benefits. Section 3 shows how to implement continuations in C++. Section 4 shows how to use macros and continuations to implement the nondeterministic abstractions. Section 5 presents the experimental results which shows that the nondeterministic abstractions can be implemented efficiently and compare well with the search implementation of GECODE.

2 The Search Abstractions

This section describes the search abstractions in C++. Section 2.1 starts by describing the nondeterministic abstractions used to define the search tree to explore. These abstractions are parameterized by a search controller that specifies how to explore the search tree. Search controllers are briefly discussed in Section 2.2 and are presented in depth in [15].

2.1 Nondeterministic Abstractions

The nondeterministic abstractions are mostly modelled after OPL [14].

Static Choices The `try` construct creates a binary search node representing the choice between two alternatives. The snippet

```
0.  TRY(sc)
1.    cout << "yes" <<endl;
2.  OR(sc)
3.    cout << "no" <<endl;
4.  ENDTRY(sc)
```

```
0.  TRYALL(<sc>, <param>, <low>, <high>, <condition>, <ordering>)
1.    [<Statement>]*
2.  ENDTRYALL(<sc>)
```

Fig. 1. The Syntax of the TRYALL Construct.

nondeterministically produces two lines of output: the first choice displays yes, while the second one displays no. When the search controller sc implements a depth-first strategy, the instruction first executes the first choice, while the second choice is executed upon backtracking.

Note that TRY, OR, and ENDTRY are not extensions to C++: they are simply macros that encapsulate the instructions to create a search node, to implement backtracking, and to close the search node. The above code thus executes on standard C++ compilers.

Dynamic Choices The TRYALL construct iterates over a range of values, filtering and ordering the candidate values dynamically. Figure 1 depicts the general syntax of the construct. The first parameter <sc> is the search controller. The <param> argument is the local variable used to store the selected value. Parameters <low> and <high> define the range of values, while <condition> holds for those values to consider in the range. Finally, the expression <ordering> specifies the order in which to try values. For instance, the snippet

```
0.  TRYALL(sc, p, 0, 5, (p%2)==0, -p)
1.    cout << "p = "<< p << endl;
2.  ENDTRYALL(sc)
```

nondeterministically produces three lines of output: p=4, p=2, and p=0. The instruction binds the parameter p to values 0 through 5 in increasing order of -p and skips those violating the condition (p%2)==0.

Encapsulated Search The EXPLOREALL construct implements an encapsulated search that initializes the search controller and produces all solutions to its body. Figure 2 illustrates an encapsulated search for implementing a simple labeling procedure. The body of the encapsulated search (lines 2–9) iterates over the values 0..2 (line 2) and nondeterministically assigns x[i] to 0 or 1 (lines 3–7). Once all the elements in array x are labeled, the array is displayed in line 9. The right part of Figure 2 depicts the output of the encapsulated search for a depth-first search controller. Other similar constructs implement encapsulated search to find one solution or to find a solution optimizing an objective function.

It is important to emphasize some benefits of the nondeterministic abstractions. First, the code freely interleaves nondeterministic abstractions and arbitrary C++ code: it does not require the definition of classes, objects, or goals. Second, the nondeterministic structure of the program is clearly apparent, simplifying debugging with traditional support from software environments. In particular, C++ debuggers can be used on these nondeterministic programs, enabling users to follow the control flow of their programs at a high level of abstraction.

```
0.  int x[3] = -1, -1, -1;                                  0,0,0
1.  EXPLOREALL(sc)                                          0,0,1
2.    for(int i=0; i<3 ; i++) {                             0,1,0
3.      TRY(sc)                                             0,1,1
4.        x[i] = 0;                                         1,0,0
5.      OR(sc)                                              1,0,1
6.        x[i] = 1;                                         1,1,0
7.      ENDTRY(sc)                                          1,1,1
8.    }
9.    cout << x[0]<<','<<x[1]<<','<<x[2]<<endl;
10. ENDEXPLOREALL(sc)
```

Fig. 2. An Example of Encapsulated Search.

```
0.  EXPLOREALL(dfs)
1.  for(int q = 0; q < N-1; q++)
2.    if (queen[q]->getSize()>1) {
3.      TRYALL(dfs, p, 0, N-1, queen[q]->hasValue(p),p)
4.        dfs->label(v,p);
5.      ENDTRYALL(dfs);
6.    }
7.  ENDEXPLOREALL(dfs);
```

Fig. 3. A Search Procedure with a Static Variable Ordering.

Ordered Iterations Since they are implemented as macros, the nondeterministic abstractions can be naturally interleaved with `C++` code. Figure 3 illustrates a simple search procedure for the n-queens problems. In the figure, the variable-ordering is static, while the value-ordering assigns first the smallest values in the domain. When the variable ordering is dynamic, it is useful to introduce a `FORALL` abstraction to avoid tedious bookkeeping by programmers. Figure 4 depicts a search procedure where the first-fail principle is used for variable selection (line 1) and where values close to the middle of the board are tried first (line 2). Apart from the syntax which is less elegant, this search procedure is at the same level of expressiveness as the equivalent search procedures in OPL [14].

2.2 Search Controllers

The nondeterministic abstractions define the search tree to explore by creating search nodes as the program executes. They are parameterized by a search controller specifying how to explore this tree. Figure 5 shows part of the interface of search controllers. The primary methods of this interface are `addNode` and `fail`, both of which are pure and virtual. The `addNode` method adds a search node to the controller, while method `fail` is called upon encountering a failure.

Figure 6 shows a specialization of the `SearchController` for depth-first search. The `addNode` and `fail` methods use a stack of search nodes. Method `addNode` pushes a node on the stack, while method `fail` pops a search node

```
0.  EXPLOREALL(dfs)
1.    FORALL(q,0,N-1,queen[q]->getSize()>1,queen[q]->getSize())
2.      TRYALL(dfs, p, 0, N-1,queen[q]->hasValue(p),abs(mid-p)){
3.        dfs->label(v,p);
4.      ENDTRYALL(dfs);
5.    ENDFORALL;
6.  ENDEXPLOREALL(dfs);
```

Fig. 4. A Search Procedure with a Dynamic Variable Ordering.

```
0.  class SearchController {
1.  protected:
2.    SearchNode _explore;
3.  public:
4.    SearchController() {}
5.    virtual void explore(SearchNode n) { _explore = n; }
6.    virtual void fail() = 0;
7.    virtual void addNode(SearchNode n) = 0;
8.    ...
9.  }
```

Fig. 5. The Interface of Search Controller (Partial Description).

from the stack and executes it. Observe that, when programmers uses predefined search controllers, they never need to manipulate search nodes or even know that they exist: It is the role of the nondeterministic abstractions to create the search nodes and to apply the appropriate methods on the search controllers.

3 Search Nodes as C++ Continuations

As in Comet [15], the nondeterministic abstractions are implemented using continuations. Since C++ does not support continuations natively, this section describes how to implement continuations in the languages itself. Recall that a continuation captures the current state of computation, i.e., the program counter, the stack, and the registers (but not the heap). Once captured, the continuation can be executed at a later time. Figure 7 illustrates continuations on a simple example: The C++ program is shown on the left of the figure and its output is shown on the right. The program computes, using continuations, the factorial of 5, 4, ..., 0. It captures a continuation in line 6 and calls `fact` with the current value of `i` (5). After printing the result (line 7), the program tests whether the `i`'s value is not smaller than 1. In this case, `i`'s value is decremented and the continuation is executed. The execution then restarts in line 7, computes the factorial of 4, and iterates the process again.

We now show how to implement continuations in C++ using `setjmp` and `longjmp`. The interface of a continuation is specified in Figure 8. Its main methods are `restore` (to restore the stack of the continuation) and `execute` to execute the continuation. Its instance variables are used to save the buffer `_target` used

```
0.  class DFS : public SearchController {
1.    Stack<SearchNode> _stack;
2.    void addNode(SearchNode n) { _stack.push(n); }
3.    void fail() {
4.      if (_stack.empty()) _explore->execute();
5.      else _stack.pop()->execute(); }
6.  }
```

Fig. 6. The Implementation of a Depth-First Search Controller (Partial Description).

```
0.  int fact(int n){                                          fact(5)=120
1.    if (n==0) return 1; return n*fact(n-1);                 fact(4)=24
2.  }                                                         fact(3)=6
3.  int main(int argc, char*argv[]){                          fact(2)=2
4.    initContinuations(&argc);                               fact(1)=1
5.    int* i = new int(5);                                    fact(0)=1
6.    Continuation* c = captureContinuation();
7.    cout <<"fact("<<*i<<")="<<fact(*i)<<endl;
8.    if(*i >=1){ (*i)--; c->execute(); }
9.    return 0;
10. }
```

Fig. 7. A Simple Example Illustrating Continuations in C++.

by `longjmp`, the stack, and the number of times the continuation has been called (`_calls`). The constructor in lines 8–12 saves the C++ stack. Note the definition of search nodes in line 19: Search nodes are simply pointers to continuations.

Figure 9 depicts the core of the implementation. Function `initContinuation` stores the base of the stack (i.e., the address of `argc`) in static variable `baseStack`. A correct value for `baseStack` is critical to save and restore continuations.

Function `captureContinuation` (line 2–11) captures a continuation. Line 5 creates an instance of class `Continuation` using, as arguments, the address of `k` and the size `baseStack-(char*)&k` to be able to save the stack. Line 6 uses the C++ instruction `setjmp` to save the program counter and the registers into the field `_target` of the continuation. After execution of line 6, the continuation has been captured and line 7 is either executed just after the capture (in which case `jmpval` is 0) or after a call to `longjmp` on `_target` (in which case `jmpval` is the continuation passed to `longjmp`). In the second case, function `captureContinuation` must restore the stack (line 8) before returning the continuation in line 9.

Method `restore` is depicted in lines 11–14. It restores the stack in line 12 and increments instance variable `_calls` to specify that the continuation has been called one more time. This last operation is important to implement the non-deterministic abstractions. Finally, method `execute` simply performs a `longjmp` on instance variable `_target`, passing the continuation itself. The effect is to restart the execution in line 7 of `captureContinuation` after having assigned `jmpval` to the continuation, inducing the stack restoration in line 8.

```
0.  class Continuation {
1.    jmp_buf _target;      // instruction to return to. used by long_jmp
3.    int _length;          // size of captured stack
4.    void* _start;         // location to restore the stack
5.    char* _data;          // copy of the stack
6.    int _calls;           // number of calls to this continuation
7.  public:
8.    Continuation(int len,void* start) {
9.      _length = len; _start = start;
10.     _data = new char[len];
11.     memcpy(_data,_start,_length);
12.   }
13.   ~Continuation();
14.   void restore();
15.   void execute();
16.   const int nbCalls() const {return _calls;}
17.   friend Continuation* captureContinuation();
18. };
19. typedef SearchNode Continuation*;
```

Fig. 8. The Interface of Continuations.

Continuations only use standard C functions and are thus portable to all architectures with a correct implementations of these functions. In absence of `setjmp` and `longjmp`, `captureContinuation` and `execute` can be implemented using `getContext` and `setContext`, or in assembly. Our implementation based on `setjmp/longjmp` has been successfully tested on three different hardware platforms (Intel x86, PowerPC, and UltraSparc) and four operating systems (Linux, Windows XP, Solaris, and OSX). The only platform where it fails is Itanium because of its implementation of `setjmp` and `longjmp` does not conform to the specifications: it does not allow several `longjmp` calls for the same `setjump`.

It is important to note that the implementation of Ilog Solver [5] also uses `setjmp` and `longjmp` [11]. The novelty here is to save the stack before calling `setjmp` and restoring the stack after calling `longjmp`. The benefits are twofold. On the one hand, it enables the implementation of high-level nondeterministic abstractions such as `tryall` in C++. On the other hand, it enables continuations to be called at any time during the execution even if the stack has fundamentally changed. As a result, continuations provide a sound basis for complex search procedures jumping from node to node arbitrarily in the search tree.

Finally, it is worth emphasizing that the nondeterministic abstractions can be used across function/method calls. They can also be encapsulated in methods to provide generic search procedures for various problem classes.

4 Implementation

It remains to show how to implement the nondeterministic abstractions in terms of continuations. As mentioned earlier, the nondeterministic abstractions are

```
0.  static char* baseStack = 0;
1.  void initContinuations(int* base) { baseStack = (char*) base; }
2.  Continuation* captureContinuation() {
3.    Continuation* jmpval;
4.    Continuation* k;
5.    k = new Continuation(baseStack-(char*)&k,&k);
6.    jmpval = (Continuation*) setjmp(k->_target);
7.    if (jmpval != 0)
8.      jmpval->restore();
9.    return k;
10. }
11. void Continuation::restore() {
12.   memcpy(_start,_data,_length);
13.   ++_calls;
14. }
15. void Continuation::execute() { longjmp(_target,(int)this); }
```

Fig. 9. Functions to create and use Continuations.

implemented as macros that capture and call continuations and apply methods of the search controller. Recall that the use of macro expansions does not interfere with the debugger. Breakpoints placed within the body of the search procedure behave as expected. Also, since a macro is expanded to a single line of code, line-by-line stepping skips over the macro, while single stepping enters the body of the macro definition.

Static Choices The statement

```
TRY(sc) ⟨ A ⟩ OR(sc) ⟨ B ⟩ ENDTRY(sc)
```

is rewritten as

```
0.  Continuation* cont = captureContinuation();
1.  if(cont->nbCalls() == 0) {
2.    sc->addNode(cont);
3.    ⟨ A ⟩
4.  } else {
5.    ⟨ B ⟩
6.  }
```

where lines 1–2 are produced by the macro `TRY(sc)`, line 4 is from the macro `OR(sc)`, and lines 6 is from `ENDTRY(sc)`. The resulting code can be explained as follows. Line 0 captures a continuation. It then tests whether the continuation has not yet been called (line 1), which is always the case the first time the `TRY` instruction is executed. As a result, lines 2–3 are executed: line 2 adds the continuation (or search node) in the search controller while line 3 executes the instructions `A`. When the continuation is called, execution comes back to line 1 but the number of calls to the continuation is not zero. As a result, the instructions `B` in line 5 are executed. Observe that the exploration strategy is left to the controller: the above implementation does not prescribe a depth-first strategy and the continuation can be called at any time during the computation to execute line 5.

Encapsulated Search The EXPLOREALL is an encapsulated search exploring the search tree specified by its body. The code

```
EXPLOREALL(sc) ⟨ A ⟩ ENDEXPLOREALL(sc)
```

is rewritten as:

```
0.  Continuation *cont = captureContinuation();
1.  if(cont->nbCalls()==0) {
2.    sc->explore(cont);
3.    ⟨A⟩
4.    sc->fail();
5.  }
```

where lines 0–2 are produced by `EXPLOREALL(sc)` and lines 4–6 are generated by `ENDEXPLOREALL(sc)`. The key implementation idea is to create a continuation `cont` representing what to do when the search tree defined by `A` is fully explored. The test in line 2 holds for the first execution of `EXPLOREALL`, in which case lines 2–4 are executed. They tell the search controller to start an encapsulated search, execute `A`, and fail (line 4). The failure makes sure that the search tree defined by `A` is fully explored. When this is the case, the continuation `cont` is executed, leading to the execution of line 6 which terminates the encapsulated search.

Dynamic Choices Consider now the `TRYALL` abstraction. The presentation is in stepwise refinements, starting first with a version with no filtering and ordering and adding these features one at a time. The code

```
TRYALL4(sc, p, low, high) ⟨ A ⟩ ENDTRYALL4(sc)
```

is rewritten as

```
0.  int p=(low);
1.  int* curIndex = new int(p);
2.  Continuation *cont = captureContinuation();
3.  if (*curIndex <= (high) ){
4.    p=(*curIndex);
5.    ++(*curIndex);
6.    (sc)->addNode(cont);
7.    ⟨ A ⟩
8.  } else {
9.    delete curIndex;
10.   sc->fail();
11. }
```

where lines 0–6 are generated by `TRYALL4(sc, p, low, high)` and lines 8–11 by `ENDTRYALL4(sc)`. There are two features to emphasize here. First, `p` is declared as a local variable in line 0 and can then be used naturally in `A`. Second, `curIndex` holds an integer representing the current index in `low..high`. `curIndex` is allocated on the heap since otherwise its value would be restored to `low` when the continuation is called. The rest of the `TRYALL` implementation then follows the `TRY` implementation. The continuation is captured in line 2. As long as the current index is within the range (line 3), a call to the continuation (or the first call to `TRYALL`) assigns the current index to `p` (line 4), increments

```
0.  int p=low;
1.  int* curIndex = new int(p);
2.  Continuation* cont = captureContinuation();
3.  bool found=false;
4.  while ((*curIndex) <= (high)) {
5.    p=(*curIndex)++;
6.    if (cond) { found=true; break; }
7.  }
8.  if (found){
9.    sc->addNode(cont);
10.   ⟨ A ⟩
11. } else {
12.   delete curIndex;
13.   sc->fail();
14. }
```

Fig. 10. The TRYALL Implementation with Filtering.

the index for subsequent iterations (line 5), adds the continuation to the search controller to allow additional iterations (line 6), and executes A (line 7). When all values in the range are explored, the implementation releases the space taken by curIndex (line 9) and fails (line 10).

Figure 10 shows how to generalize the implementation when values in the range are filtered as in

TRYALL5(sc, p, low, high, cond) ⟨ *A* ⟩ ENDTRYALL5(sc)

The main novelty in Figure 10 is the addition of a loop (lines 4–6) to find the first element in the range satisfying the condition cond.

Finally, Figure 11 depicts the implementation of the TRYALL abstraction with filtering and ordering, i.e.,

TRYALL(sc, p, low, high, cond, ordering) ⟨ *A* ⟩ ENDTRYALL(sc)

The key idea is to replace curIndex by an array of Booleans that keep track of which values have been tried already. The array is allocated on the heap in line 2 and initialized in line 3. The continuation is captured in line 4 and the rest of the code depicts the treatment performed for all successive calls to the TRYALL instruction. Lines 7–8 search for the available element in the range satisfying condition cond and minimizing expression ordering. If such an element exists (line 13), the continuation is added to the controller, array avail is updated, and the instructions A are executed. Otherwise, all elements have been tried, which means that array avail can be released and the TRYALL must fail.

Iterations with Ordering The implementation of the FORALL instruction is simpler since it does not create search nodes: it simply iterates over the range and selects the value satisfying the condition and minimizing the ordering expression. Figure 12 describes the implementation of the code

FORALL(sc, p, low, high, cond, ordering) ⟨ *A* ⟩ ENDFORALL(sc)

Unlike TRYALL, the array avail must be allocated on the C++ stack (line 2). Indeed, the available values must be restored on backtracking, which is achieved automatically by continuations when the array is allocated on the C++ stack.

```
0.  int p = 0;
1.  int l = (low); int h = (high);
2.  bool* avail = new bool[h-l+1];
3.  for(int i=0; i < h-l+1; i++) avail[i]=true;
4.  Continuation *cont = captureContinuation();
5.  bool found=false;
6.  int bestEval = INT_MAX;
7.  for(int k=l; k <= h; k++)
8.    if( (bestEval > (ordering)) && avail[k-l] && (cond)) {
9.      found = true;
10.     bestEval = (ordering);
11.     p = k;
12.   }
13. if (found) {
14.   avail[p-l]=false;
15.   sc->addNode(cont);
16.   ⟨ A ⟩
17. } else {
18.   delete[] avail;
19.   sc->fail();
20. }
```

Fig. 11. The Implementation of the `TRYALL` Abtraction with Filtering and Ordering.

5 Experimental Results

This section presents the experimental results demonstrating the efficiency of the implementation. It first shows that the cost of using continuations is not prohibitive. Then, it demonstrates that the abstractions are comparable in efficiency to the search procedures of existing constraint libraries. The CPU Times are given on a Pentium IV 2.0 GHz running Linux 2.6.11.

On the Efficiency of Continuations One possible source of inefficiency for the nondeterministic abstractions is the overhead of capturing and restoring continuations. To quantify this cost, we use a simple backtrack search for the queens problem and we compare a search procedure written in C++ (and thus with a recursive style) with a search procedure using the nondeterministic abstractions (and thus with an iterative style). Figures 13 and 14 depict the two search procedures and their common `attack` function. Table 1 shows the runtime of the recursive (R) and nondeterministic (N) search procedures and the percentage increase in CPU time. The results show that the percentage increase in CPU time decreases as the problem size grows and goes down to 54% for the 32-queens problem. These results are noteworthy, since they use a mainstream, non-garbage collected, highly efficient language. Moreover, these programs do not involve any constraint propagation and do not need to save and restore the states of domain variables and constraints. As such, these tests represents the pure cost of the abstractions compared to the hand-coded implementation.

```
0.  int p = 0;
1.  int l = (low); int h = (high);
2.  bool* avail = (bool*) alloca(h-l+1);
3.  for(int i=0; i < h-l+1; i++) avail[i]=true;
4.  while(true) {
5.    bool found=false;
6.    int bestEval = INT_MAX;
7.    for(int k=l; k <= h; k++)
8.      if( (bestEval > (ordering)) && avail[k-l] && (cond)) {
9.        found = true;
10.       bestEval = (ordering);
11.       p = k;
12.     }
13.   if (found) {
14.     avail[p-l]=false;
15.     ⟨ A ⟩
16.   } else break;
17. }
```

Fig. 12. The Implementation of the `FORALL` Abtraction with Filtering and Ordering.

n	16	18	20	22	24	26	28	30	32
R	.007	.028	.152	1.486	.405	.443	3.748	78.900	133.86
N	.014	.063	.308	2.878	.731	.762	6.175	125.64	205.82
$(N-R)/R$	1.00	1.25	1.026	0.937	0.805	0.720	0.648	0.592	0.538

Table 1. The Pure Cost of the Nondeterministic Abstractions

Comparison with an Existing Library We now compare the efficiency of the nondeterministic abstractions with an existing `C++` constraint programming library: Gecode [4]. Figure 15 shows the partial Gecode model used for the queens problem. The depth-first search in Gecode has a *copy distance* parameter specifying the number of branchings to perform using recomputation before cloning the search space. A value of 1 (meaning no recomputation) was used in the experiments as that tends to give the best performance for the queens problem. Observe the call to the built-in search procedure in line 11: It has a static left-to-right ordering and starts first with the smallest values. We compare the performance of Gecode with a similar statement using a constraint programming library using trailing and the search procedure depicted in Figure 3 and the search controller (partially) described in Figure 16. Both implementations use exactly the same search procedure: it is built-in in the case of Gecode but uses our high-level nondeterministic abstractions in our case. Note also that the search controller now calls the CP manager to push and pop choices (in addition to search nodes) to implement trailing. Table 2 depicts the computational results: they indicate that the program with the nondeterministic abstractions is slightly more efficient than Gecode. These results tend to demonstrate the

```
bool attack(int* queen,int n,int i,int v){
  for(int k=0; k < i; k++) {
    if( queen[k] == v )return true;
    if( queen[k]+k == v+i)return true;
    if( queen[k]-k == v-i)return true;
  }
  return false;
}
bool search(int* queen,int n,int i){
  if(i >= n) return true;
  for(int v = 0; v < n; v++){
    if (attack(queen,n,i,v)) continue;
    queen[i]=v;
    if (search(queen,n,i+1)) return true;
  }
  return false;
}
search(queens,N,0);
```

Fig. 13. The Recursive Version of the Simple Backtrack Search.

```
for(int q=0;q<N;q++){
  TRYALL4(sc, v, 0, N-1, !attack(queen,N,q,v))
    queen[q]=v;
  ENDTRYALL4(sc);
}
```

Fig. 14. The Nondeterministic Version of the Simple Backtrack Search.

practicability of our approach, since the queens problem has little propagation compared with realistic constraint programs and hence the cost of the abstractions will be even more negligible on more complex applications.

Programming Search Engines The previous comparison used two different solvers and the difference in efficiency may partially be attributed to their respective efficiency. To overcome this limitation, our last experiment only uses Gecode as the underlying solver. It compares the built-in implementation of depth-first search in Gecode with an implementation using our nondeterministic abstractions. Recall that Gecode manipulates computation spaces representing the search tree. The following C++ code

```
0.  EXPLORE(gecode)
1.    while (gecode->needBranching()) {
2.      int alt = gecode->getNbAlternatives();
3.      TRYALL4(gecode, a, 0, alt-1)
4.        gecode->tryCommit(a);
5.      ENDTRYALL4(gecode);
6.  }
7.  ENDEXPLORE(gecode);
```

```
0.  class Queens : public Example {
1.      IntVarArray q; // Position of queens on boards
2.    public:           // The actual problem
3.      Queens(const Options& opt) : q(this,opt.size,0,opt.size-1) {
4.        const int n = q.size();
5.        for (int i = 0; i<n; i++)
6.          for (int j = i+1; j<n; j++) {
7.            post(this, q[i] != q[j]);
8.            post(this, q[i]+i != q[j]+j);
9.            post(this, q[i]-i != q[j]-j);
10.         }
11.       branch(this, q, BVAR_NONE, BVAL_MIN);
12.     }
13.   }
14.   Example::run<Queens,DFS>(opt);
```

Fig. 15. The GECODE Model for the Queens Problem.

```
0.  class CPDFS : public SearchController {
1.    Stack<SearchNode> _stack;
2.    CPManager* _mgr;
3.  public:
4.    ...
5.    void addNode(SearchNode n) {
6.      _nodes->push(n);
7.      _mgr.pushChoice(f)
8.    }
9.    void fail() {
10.     _mgr->popChoice();
11.     if (_stack.empty()) _explore->execute();
12.     else _stack.pop()->execute();
13.   }
14. }
```

Fig. 16. A Depth-First Search Controller for a CP Library with Trailing.

is an implementation of depth-first search for GECODE that uses our nondeterministic abstraction. The code iterates branching until the tree is fully explored (line 1). To branch, the code retrieves the number of alternatives (line 2) and performs a `TRYALL` to try each alternative (line 4). The code uses a `gecode` controller to clone and restore the spaces appropriately (which is not shown for space reasons). Note also the combination of a `C++ while` instruction with `TRYALL`.

Table 3 depicts the computational results. This evaluation has the merit of comparing the search procedures with exactly the same constraint solver and the search procedure coded by the designer of the library. Once again, the nondeterministic abstractions are slightly more efficient than the builtin implementation of GECODE, although they perform exactly the same number of clones, failures, and propagation calls. The results thus indicate that the nondeterministic abstractions are not only expressive and natural; they are also very efficient.

n	16	18	20	22	24	26	28	30	32
Gecode	.06	.20	.98	7.83	2.08	2.08	16.34	301.27	507.08
ND	.05	.18	.92	7.56	1.93	1.83	14.96	294.20	498.37

Table 2. Performance Comparison (in Seconds) with GECODE on the Queens Problem.

n	16	18	20	22	24	26	28	30	32
Gecode	.06	.20	.98	7.83	2.08	2.08	16.34	301.27	507.08
$ND + Gecode$	0.05	.19	.97	7.55	2.02	1.96	16.28	292.80	495.69

Table 3. Performance Comparison in Seconds on GECODE Only.

6 Conclusion

This paper showed how to use macros and continuations in C++ to support the high-level nondeterministic abstractions found in recent constraint languages. The resulting design has several benefits. The abstractions promote an iterative style for search procedures, simplify debugging since the C++ stack now reflects directly the control flow of the program, and allow for natural implementations of search strategies. The implementation, which uses `setjmp/longjmp`, is shown to compare well with the C++ library GECODE.

Acknowledgments

This work was supported by the National Science Foundation grant DMI-0423607. Special thanks to the third reviewer for some insightful comments.

References

1. A. Colmerauer. Opening the Prolog-III Universe. *BYTE Magazine*, 12(9), 1987.
2. de Givry, S. and Jeannin, L. Tools: A library for partial and hybrid search methods. In *CP-AI-OR'03*.
3. N.C. Heintze, S. Michaylov, and P.J. Stuckey. CLP($\Re$) and some Electrical Engineering Problems. In *ICLP-87*.
4. http://www.gecode.org/. Generic Constraint Development Environment, 2005.
5. Ilog Solver 4.4. Reference Manual. Ilog SA, Gentilly, France, 1998.
6. F. Laburthe and Y. Caseau. SALSA: A Language for Search Algorithms. In *CP'98*.
7. J-L. Lauriere. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.
8. L. Michel and P. Van Hentenryck. Modeler++: A Modeling Layer for Constraint Programming Libraries. In *CP-AI-OR'2001*.
9. L. Perron. Search Procedures and Parallelism in Constraint Programming. In *CP'99*.
10. J-F. Puget. A C++ Implementation of CLP. In *Proceedings of SPICIS'94*.
11. J.-F.. Puget. Personal Communication, March 2006.
12. C. Schulte. Programming Constraint Inference Engines. In *CP'97*.
13. P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. The MIT Press, Cambridge, MA, 1989.
14. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, Cambridge, Mass., 1999.
15. P. Van Hentenryck and L. Michel. Nondeterministic Control for Hybrid Search. In *CP-AI-OR'05*.