



# Localizer

LAURENT MICHEL  
*Brown University, Box 1910, Providence, RI 02912*

ldm@cs.brown.edu

PASCAL VAN HENTENRYCK  
*Brown University, Box 1910, Providence, RI 02912*

pvh@cs.brown.edu

**Abstract.** Local search is a traditional technique to solve combinatorial search problems which has raised much interest in recent years. The design and implementation of local search algorithms is not an easy task in general and may require considerable experimentation and programming effort. However, contrary to global search, little support is available to assist the design and implementation of local search algorithms. This paper describes the design and implementation of LOCALIZER, a modeling language for implementing local search algorithms. LOCALIZER makes it possible to express local search algorithms in a notation close to their informal descriptions in scientific papers. Experimental results on Boolean satisfiability, graph coloring, graph partitioning, and job-shop scheduling show the feasibility of the approach.

**Keywords:** modeling language, local search, incremental algorithms

## 1. Introduction

Most combinatorial search problems are solved through global or local search. In global search (or systematic search), a problem is divided into subproblems until the subproblems are simple enough to be solved directly. In local search, an initial configuration is generated and the algorithm moves from the current configuration to a neighboring configuration until a solution (decision problems) or a good solution (optimization problems) has been found or the resources available are exhausted. The two approaches have complementary strengths, weaknesses, and application areas. The design of global search algorithms is now supported by a variety of tools, ranging from modeling languages such as AMPL [7] and OPL [25] to constraint programming languages such as CHIP [6], ILOG SOLVER [19], CLP( $\mathcal{R}$ ) [9], PROLOG-III [4], and OZ [8] to name only a few. In contrast, little attention has been devoted to the support of local search, despite the increasing interest in these algorithms in recent years. (Note however there are various efforts to integrate local search in CLP languages, e.g., [23]). The design of local search algorithms is not an easy task however. The same problem can be modeled in many different ways (see for instance [11]), making the design process an inherently experimental enterprise. In addition, efficient implementations of local search algorithms often require maintaining complex data structures incrementally, which is a tedious and error-prone activity.

LOCALIZER [13] is a modeling language for the implementation of local search algorithms, combining aspects of declarative and imperative programming, since both are important in local search algorithms. LOCALIZER makes it possible to write local search algorithms in a notation close to the informal presentation found in scientific publications, while inducing a reasonable overhead over special-purpose implementations. LOCALIZER offers support for defining traditional concepts like neighborhoods, acceptance criteria, and restarting states.

```

5      if value(s) < bestBound then
5.1      bestBound := value(s);
5.2      best := s;

```

```

procedure LOCALIZER
begin
1 s := startState();
2 for search := 1 to MaxSearches while Global Condition do
3   for trial := 1 to MaxTrials while Local Condition do
4     if satisfiable(s) then
5       return s;
6     select n in neighborhood(s);
7     if acceptable(n) then
8       s := n;
9   s := restartState(s);
end

```

Figure 1. The computation model of localizer.

In addition, LOCALIZER also introduces the declarative concept of *invariants* in order to automate the most tedious and error-prone aspect of local search procedures: incremental data structures. Invariants provide a declarative way to specify what needs to be maintained to define the neighborhood and the objective function.

This paper is a progress report describing the status of LOCALIZER as of February 1998. Its main focus is on the language and its implementation.<sup>1</sup> It is not intended as a final word on the language, since new, higher-level, extensions are currently under evaluation. The paper however describes the core of LOCALIZER which will probably not evolve in significant ways. The paper is organized in four main parts. Section 2 gives readers a quick tour of LOCALIZER. Section 3 describes the language in more detail. Section 4 describes the implementation of invariants which are the cornerstone of LOCALIZER. Section 5 summarizes some experimental results from several applications. Section 6 discusses related work. Finally, Section 7 concludes the paper.

## 2. A Tour of LOCALIZER

This section gives an overview of the main features of LOCALIZER. It starts by reviewing the computational model of LOCALIZER and the general form of LOCALIZER statements. It then considers the two main contributions of LOCALIZER: invariants and neighborhoods.

### 2.1. The Computation Model

To understand statements in LOCALIZER, it is best to consider the underlying computational model first. Figure 1 depicts the computational model of LOCALIZER for decision problems.

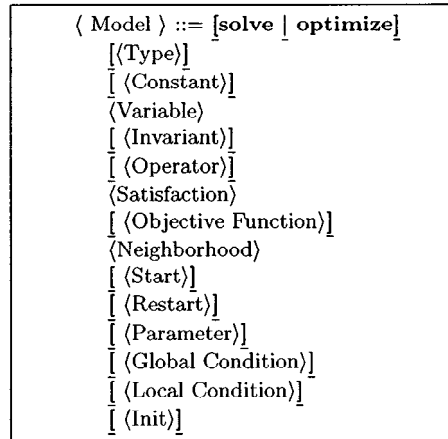


Figure 2. The structure of LOCALIZER statements.

The model captures the essence of most local search algorithms. The algorithm performs a number of local searches (up to **MaxSearches** and while a global condition is satisfied). Each local search consists of a number of iterations (up to **MaxTrials** and while a local condition is satisfied). For each iteration, the algorithm first tests if the state is satisfiable, in which case a solution has been found. Otherwise, it selects a candidate move in the neighborhood and moves to this new state if this is acceptable. If no solution is found after **MaxTrials** or when the local condition is false, the algorithm restarts a new local iteration in the state *restartState(s)*. The computation model for optimization problems is similar, except that line 5 needs to update the best solution so far if necessary, e.g., in the case of a minimization,

5	<b>if</b> <i>value(s)</i> < <b>bestBound</b> <b>then</b>
5.1	<b>bestBound</b> := <i>value(s)</i> ;
5.2	<i>best</i> := <i>s</i> ;

The optimization algorithm of course should initialize **bestBound** properly and return the best solution found at the end of the computation.

## 2.2. The Structure of LOCALIZER Statements

The purpose of a LOCALIZER statement is to specify, for the problem at hand, the instance data, the state, and the generic parts of the computation model (e.g., the neighborhood and the acceptance criterion). A LOCALIZER statement consists of a number of sections as depicted in Figure 2. The instance data is defined by the **Type**, **Constant**, and **Init** sections, using traditional data structures from programming languages. The state is defined as the values of the variables. The neighborhood is defined in the **Neighborhood** section, using objects from previous sections. The acceptance criterion is part of the definition of the

neighborhood. The initial state is defined in section **Start**. The restarting states are defined in section **Restart**, the parameters (e.g. **MaxTrials**) are given in the **Parameter** section, and the global and local conditions are given in sections **Global Condition** and **Local Condition**. Note that all the identifiers in boldface in the description of computation model (e.g., **search** and **trial**), are in fact keywords of LOCALIZER.

As mentioned previously, the most original aspects of LOCALIZER are in the specifications of the neighborhood and the acceptance criterion. Of course, some of the notations are reminiscent of languages such as AMPL [7] and CLAIRE [3] at the syntactical level but the underlying concepts are fundamentally different. In the rest of this section, we describe the most original aspects of LOCALIZER without trying to be comprehensive.

### 2.3. *The Running Example*

This overview mostly uses Boolean satisfiability to illustrate the concepts of LOCALIZER. A Boolean satisfiability problem amounts to finding a truth assignment for a propositional formula in conjunctive normal form. The input is given as a set of clauses, each clause consisting of a number of positive and negative literals. As is traditional, a literal is simply an atom (positive atom) or the negation of an atom (negative atom). A clause is satisfied as soon as at least one of its positive atoms is true or at least one of its negative atoms is false. The local search statement considered for Boolean satisfiability is based on the GSAT algorithm by Selman et al. in [22], where a local search move consists of flipping the truth value of an atom.

A local improvement statement for Boolean satisfiability is described in Figure 3. In the statement, atoms are represented by integers 1 to  $n$  and a clause is represented by two sets: the set of its positive atoms  $p$  and the set of its negative atoms  $n$ . This data representation is specified in the **Type** section. A problem instance is specified by an array of  $m$  clauses over  $n$  variables. The instance data is declared in the **Constant** section and initialized in the **Init** section which is not shown. The state is specified by the truth values of the atoms and is captured in the array  $a$  of variables in the **Variable** section. Variable  $a[i]$  represents the truth value of atom  $i$ . The **Invariant** section is the key component of all LOCALIZER statements: it describes, in a declarative way, the data structures which must be maintained incrementally. Invariants are reviewed in detail in Section 2.4. In the statement depicted in Figure 3, they maintain the number of true literals  $nbtl[c]$  in each clause  $c$  and the number of satisfied clauses  $nbClauseSat$ . The **Satisfiable** section describes when the state is a solution (all clauses are satisfied), while the **Objective Function** section describes the objective function (maximize the number of satisfied clauses) used to drive the search. The **Neighborhood** section describes the actual neighborhood and the acceptance criterion. The neighborhood consists of all the states which can be obtained by flipping the truth value of an atom and a move is accepted if it improves the value of the objective function. The **Neighborhood** section is another important part of LOCALIZER and is reviewed in more detail in Section 2.5. The **Start** and **Restart** sections describe how to generate an initial state and a new state when restarting a search. They both use a simple random generation in the statement.

It is interesting at this point to stress the simplicity of the statement, since it is difficult to imagine a more concise formal statement of the algorithm.

```

Solve
Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  cl: array[1..m] of clause = ...;
Variable:
  a: array[1..n] of boolean;
Invariant:
  nbtI : array[ i in 1..m] of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  nbClauseSat : int = sum(i in 1..m) (nbtI[i] > 0);
Satisfiable:
  nbClauseSat = m;
Objective Function:
  maximize nbClauseSat;
Neighborhood:
  move a[i] := !a[i]
  where i from {1..n}
  accept when improvement;
Start:
  forall(i in 1..n) a[i] := random({true,false});
Restart:
  forall(i in 1..n) a[i] := random({true,false});

```

Figure 3. A local improvement statement for Boolean satisfiability.

## 2.4. Invariants

Invariants are probably the most important tool offered by LOCALIZER to support the design of local search algorithms. They make it possible to specify **what** needs to be maintained incrementally without considering **how** to do so. Informally speaking, an invariant is an expression of the form  $v : t = exp$  and LOCALIZER guarantees that, at any time during the computation, the value of variable  $v$  of type  $t$  is the value of the expression  $exp$  (also of type  $t$  of course). For instance, the invariant

$$nbtI : \text{array}[ i \text{ in } 1..m] \text{ of int} = \text{sum}(i \text{ in } cl[i].p) a[j] + \text{sum}(j \text{ in } cl[i].n) !a[j];$$

in the Boolean satisfiability statement specifies that  $nbtI[c]$  is equal to the sum of all true positive atoms and all false negative atoms in clause  $c$ , for all clauses in  $1..m$ . LOCALIZER uses efficient incremental algorithms to maintain these invariants during the computation, automating one of the tedious and time-consuming tasks of local search algorithms. For instance, whenever a value  $a[k]$  is changed,  $nbtI[c]$  is updated in constant time.

LOCALIZER allows a wide variety of invariants over complex data structures. The invariant (also from the Boolean satisfiability statement)

$$nbClauseSat : \text{int} = \text{sum}(i \text{ in } 1..m) (nbtI[i] > 0);$$

illustrates the use of relations inside an invariant. A relation, when used inside an expression, is considered a 0-1 integer, i.e., the relation evaluates to 1 when true and 0 otherwise. The excerpt

```

C      : array[1..n] of int = distribute(x,1..n,1..n);
Empty  : {int} = { i : int | select i from 1..n where size(C[i]) = 0 };
NEmpty : {int} = { i : int | select i from 1..n where size(C[i]) > 0 };
unused : int = minof(Empty);
Candidates : {int} = NEmpty union unused;
B      : array[k in 1..n] of {edge} = {(i, j) : edge | select i from C[k] &
                                     select j from C[k]
                                     where A[i, j] };
f      : int = sum(i in 1..n) (2 * size(C[i]) * size(B[i]) - size(C[i])2);
countB : int = sum(i in 1..n) size(B[i]);

```

is taken from a graph-coloring statement implementing an algorithm in [11]. The graph-coloring problem amounts to finding the smallest number of colors to label a graph such that adjacent vertices have different colors. For a graph with  $n$  vertices, the algorithm considers  $n$  colors which are the integers between 1 and  $n$ . Color class  $C_i$  is the set of all vertices colored with  $i$  and the bad edges of  $C_i$ , denoted by  $B_i$ , are the edges whose vertices are both colored with  $i$ . The main idea of the algorithm is to minimize the objective function  $\sum_{i=1}^n 2|B_i||C_i| - |C_i|^2$  whose local minima are valid colorings. To minimize the function, the algorithm chooses a vertex and chooses a color whose color class is non-empty or one of the unused colors. It is important to consider only one of the unused colors to avoid a bias towards unused colors. The invariant

```

B: array[k in 1..n] of {edge} =
  {(i, j) : edge | select i from C[k] & select j from C[k] where Adj[i, j] };

```

expresses that  $B[k]$  is the set of edges obtained by selecting two adjacent vertices in color class  $k$ . It illustrates that LOCALIZER can maintain queries over sets varying in time (since  $C[k]$  evolves during the local search). The invariant

```

C: array[1..n] of int = distribute(x,1..n,1..n);

```

is equivalent to, but more efficient than,

```

C: array[i in 1..n] of int = { j | select i from 1..n where x[j] = i };

```

This primitive function is provided, since it is useful in a large variety of applications. The invariant

```

NEmpty : {int} = { i : int | select i from 1..n where size(C[i]) > 0 };

```

defines the non-empty classes. Note that here the set  $1..n$  does not vary but the condition  $\text{size}(C[i]) > 0$  evolves over time.

Job-shop scheduling illustrates the expressiveness of invariants even further. The problem consists of a set of jobs, where each job is a sequence of tasks (e.g., the first task in a job

must precede the second task and so on) and each task executes on a given machine. Tasks executing on the same machine cannot overlap in time. The job-shop scheduling problem amounts to finding an assignment of starting dates to the tasks satisfying the precedence and non-overlapping constraints and minimizing the makespan, i.e., the maximum duration of all jobs. It is well-known that solving a job-shop problem mostly consists of determining an optimal ordering for the tasks on the various machines.

Most neighborhoods for job-shop scheduling swap only tasks on the critical path, i.e., tasks with no slack. The invariants

<pre> R : array[i in 0..N] of int := if i = 0                                then 0                                else max(R[PJ[i]] + D[PJ[i]], R[PM[i]] + D[PM[i]]); Q : array[i in 1..N + 1] of int := if i = N + 1                                then 0                                else max(Q[SJ[i]] + D[i], Q[SM[i]] + D[i]); T : array[i in 1..N] of int := R[i] + Q[i]; M : int := max(j in 1..N) T[j]; C : {int} := {i : int   select i from 1..N where PM[i] &lt;&gt; 0 and                                R[PM[i]] + D[PM[i]] = R[i] and                                T[i] = M}; </pre>
--

can be used to compute these critical tasks. Invariant  $R[t]$  maintains the release date of a task  $t$  (i.e., the earliest date at which it can start) while invariant  $Q[t]$  maintains the tail of task  $t$  (i.e., the longest path from  $t$  to the end of the project). Invariant  $T[t]$  maintains the longest path from the start of the project to the end going through task  $t$ . Invariant  $C$  maintain the critical path. Note that  $PJ[t]$  and  $PM[t]$  denote the predecessor and the successor of task  $t$  in its job and on its machine respectively. The array  $PJ$  is static (i.e., it does not change during the computation) while array  $PM$  is dynamic since the goal of the application is to find an ordering for the machines. Similar considerations apply to  $SJ$  and  $SM$  which define successors instead of predecessors.

The power of these invariants lies in the fact that they are defined recursively and, even more important, that variables (e.g.,  $PM[i]$ ) are used to index other variables (e.g.,  $R$ ). These functionalities make LOCALIZER more expressive than traditional finite differencing techniques [16].

Once again, it is important to emphasize the significant support provided by LOCALIZER with invariants. These invariants maintain complex data structures incrementally, but users only have to specify them in a declarative way.

## 2.5. The Neighborhood

Many strategies such as local improvement, simulated annealing, and tabu search have been proposed in the last decades for local search algorithms. This section reviews how they are modeled in the **neighborhood** section, which is the other fundamental tool provided by LOCALIZER.

### 2.5.1. Local Improvement and Local Nondegradation

The statement depicted in Figure 3 uses a stochastic local improvement approach. The neighborhood section

```

Neighborhood:
  move  $a[i] := !a[i]$ 
  where  $i$  from  $\{1..n\}$ 
  accept when improvement;
```

specifies the following strategy: select a value  $i$  in  $1..n$  (i.e., select an atom), flip  $a[i]$ , and, if the resulting state improves the value of the objective function, take the move. If the state does not improve the value of the objective function, the move is not taken and LOCALIZER proceeds to the next iteration of the innermost loop in the computational model. This strategy illustrates the structure of the neighborhood definition in LOCALIZER. The **move** part specifies a state transformation: it uses traditional imperative constructs to show how to transform a state into another state. The **where** part specifies the set of objects used to specify the state transformation. The **accept** part describes when to accept the move.

The local improvement strategy can be made greedy by adding the keyword **best** in front of the move instruction, as in

```

Neighborhood:
  best move  $a[i] := !a[i]$ 
  where  $i$  from  $\{1..n\}$ 
  accept when improvement;
```

This excerpt specifies the following strategy: consider each value  $i$  in  $1..n$  which, when flipped, produces an improvement and select the one with the best improvement. Note that this strategy explores the neighborhood in a systematic way, while the previous strategy was selecting a random move and testing it for improvement. The neighborhood section

```

Neighborhood:
  first move  $a[i] := !a[i]$ 
  where  $i$  from  $\{1..n\}$ 
  accept when improvement;
```

is another approach that explores the neighborhood systematically until a move improving the value of the objective function is found. It should be contrasted with the random walk strategy presented previously.

Sometimes it is important to allow more flexibility in the local search and to allow moves which may not improve the objective function. The neighborhood

```

Neighborhood:
  best move  $a[i] := !a[i]$ 
  where  $i$  from  $\{1..n\}$ 
  accept when noDecrease;
```

accepts the best move which does not decrease the value of the objective function. The resulting statement, depicted in Figure 4, captures the essence of the GSAT algorithm.

```

Solve
Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  cl: array[1..m] of clause = ...;
Variable:
  a: array[1..n] of boolean;
Invariant:
  nbtl : array[ i in 1..m] of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  nbClauseSat : int = sum(i in 1..m) (nbtl[i] > 0);
Satisfiable:
  nbClauseSat = m;
Objective Function:
  maximize nbClauseSat;
Neighborhood:
  best move a[i] := !a[i]
  where i from {1..n}
  accept when noDecrease;
Start:
  forall(i in 1..n) a[i] := random({true,false});
Restart:
  forall(i in 1..n) a[i] := random({true,false});

```

Figure 4. A GSAT-based statement for Boolean satisfiability.

### 2.5.2. Simulated Annealing

Simulated annealing is a well-known stochastic strategy to enhance a local improvement search. It is easily expressed by the LOCALIZER neighborhood

```

Neighborhood:
  move a[i] := !a[i]
  where i from {1..n}
  accept
    when improvement → ch++;
    cor noDecrease;
    cor Pr(exp(-delta/t)) : always → ch++;

```

The key novelty here is that the accept statement may have a number of acceptance conditions which are tried in sequence until one succeeds or all fail. In addition, each acceptance condition can be associated with an action. The simulated annealing neighborhood specifies that a move is accepted when it improves the objective function, when it does not decrease the objective function, or with the standard probability of simulated annealing, which depends on a temperature parameter and the variation **delta** of the objective function. Note that the variable *ch* is incremented when there is an improvement or a decrease in the objective function. The complete statement is given in Figure 5. The statement illustrates also several new features of LOCALIZER. The **Operator** section describes two procedures which are used

```

Solve
Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  cl: array[1..m] of clause = ...;
Variable:
  a: array[1..n] of boolean;
  t: real;
  ch: int;
Invariant:
  nbtl: array[ i in 1..m ] of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  nbClauseSat : int = sum(i in 1..m) (nbtl[i] > 0);
Operator:
  void genState() {
    forall(i in 1..n) x[i] := random({true,false});
    t := 2.0;
    ch := 0;
  }
  void lowTemp() {
    t := t * 0.95;
    ch := 0;
  }
Satisfiable:
  nbClauseSat = m;
Objective Function:
  maximize nbClauseSat;
Neighborhood:
  move a[i] := !a[i]
  where i from {1..n}
  accept
  when improvement → ch++;
  cor noDecrease
  cor Pr(exp(-delta/t)) : always → ch++;
Start:
  genState();
Restart:
  lowTemp();
local condition: ch < n;

```

Figure 5. A simulated annealing statement for SAT.

subsequently in the **Start** and **Restart** sections. Operators in LOCALIZER uses traditional constructs from imperative programming languages (e.g., loops and conditions), as well as some new primitives for randomization. These features are once again described in more detail in Section 3. Note also the variables  $t$  (the temperature) and  $ch$  (the change counter) which are used in various places in the statement. The role of  $ch$  is to prevent the algorithm from spending too much time at high temperatures where moves are likely to be accepted. The local condition forces the algorithm to terminate the innermost loop of the computation model.

### 2.5.3. Tabu Search

Tabu search is another strategy to escape local optima which, in contrast to simulated annealing, does not resort to stochastic moves. The neighborhood

```

Neighborhood:
best move  $a[i] := !a[i]$ 
where  $i$  from  $\{1..n\}$  such that  $!tabu(i)$ 
accept when always  $\rightarrow t[v] := trial;$ 

```

indicates how a simple tabu search can be expressed in LOCALIZER. The key idea here is to select an atom which is not tabu. The **where** clause is generalized to include this condition. All the moves so-defined are accepted and the statement also keeps track of when an atom was last flipped by using the keyword **trial**. An atom is then tabu if it has been flipped recently, which can be expressed as

```

boolean tabu( $idx$  : int)
{
  return  $t[idx] > trial - tl;$ 
}

```

where  $tl$  is a parameter specifying the time an atom stays on the tabu list. The complete statement is described in Figure 6. Of course, more complicated tabu search algorithms (e.g., using aspiration criteria to overwrite the tabu status or a tabu list whose size varies over time) can be implemented easily.

### 2.5.4. Composing Neighborhoods

LOCALIZER makes it also possible to compose neighborhoods. For instance, the following neighborhood

```

try
  0.1:
    move
       $a[i] := !a[i]$ 
    where
       $i$  from OccurInUnsatClause
    accept when always ...;
  default:
    best move
       $a[i] := !a[i]$ 
    where
       $i$  from  $\{1..n\}$ 
    accept when noDecrease;
end

```

```

Solve
Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  cl: array[1..m] of clause = ...;
Variable:
  a: array[1..n] of boolean;
  t: array[1..n] of int;
  tl: int;
Invariant:
  nbtl : array{ i in 1..m } of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  nbClauseSat : int = sum(i in 1..m) (nbtl[i] > 0);
Operator:
  void genState() {
    tl := 10;
    forall(i in 1..n) x[i] := random({true,false});
    forall(i in 1..n) t[i] := -tl;
  }
  boolean tabu(idx : int) {
    return t[idx] > trial - tl;
  }
Satisfiable:
  nbClauseSat = m;
Objective Function:
  maximize nbClauseSat;
Neighborhood:
  best move a[i] := !a[i]
  where i from {1..n} such that !tabu(i)
  accept when always → t[v] := trial;
Start:
  genState();
Restart:
  genState();

```

Figure 6. A Tabu search statement for Boolean satisfiability.

implements the random walk/noise strategy of GSAT. Here, LOCALIZER flips an arbitrary variable in an unsatisfied clause with a probability of 0.1 and applies the standard strategy with a probability of 0.9. Note that LOCALIZER simply goes to the next iteration if the selected neighborhood is empty, since other neighborhoods may be non-empty.

### 2.5.5. Incrementality Issues

In the statements presented so far, LOCALIZER needs to simulate the move to find out how the objective function evolves. This simulation can become very expensive when few moves are accepted. In practice, local search implementations often try to evaluate the impact of the move in the current state. LOCALIZER supports this practice by allowing to specify

acceptance criteria which are evaluated in the current state. For instance, the neighborhood definition

**Neighborhood:**  
**first move**  $a[i] := !a[i]$   
**where**  $i$  **from**  $\{1..n\}$   
**accept when in current state**  
 $gain[i] \geq 0$ ;

evaluates the condition  $gain[i] \geq 0$  in the current state to determine whether to take the move. Of course, this requires to generalize the invariants to maintain  $gain[i]$  incrementally. The invariants now become

$nbtl$ : **array**  $[ i \text{ in } 1..m ]$  of **int** =  $\text{sum}(i \text{ in } cl[i].p) a[j] + \text{sum}(j \text{ in } cl[i].n) !a[j]$ ;  
 $g01$ : **array**  $[ i \text{ in } 1..n ]$  of **int** =  
 $\text{sum}(j \text{ in } po[i]) (nbtl[j] = 0) - \text{sum}(j \text{ in } no[i]) (nbtl[j] = 1)$ ;  
 $g10$ : **array**  $[ i \text{ in } 1..n ]$  of **int** =  
 $\text{sum}(j \text{ in } no[i]) (nbtl[j] = 0) - \text{sum}(j \text{ in } po[i]) (nbtl[j] = 1)$ ;  
 $gain$ : **array**  $[ i \text{ in } 1..n ]$  of **int** = **if**  $a[i]$  **then**  $g10[i]$  **else**  $g01[i]$ ;  
 $nbClauseSat$ : **int** =  $\text{sum}(i \text{ in } 1..m) (nbtl[i] > 0)$ ;

The informal meaning of the new invariants are the following.  $g01[i]$  represents the change in the number of satisfied clauses when changing the value of atom  $i$  from false to true, assuming that atom  $i$  is currently false. Obviously, the flip produces a gain for all unsatisfied clauses where atom  $i$  appears positively. It also produces a loss for all clauses where  $i$  appears negatively and is the only atom responsible for the satisfaction of the clause.  $g10[i]$  represents the change in satisfied clauses when changing the value of atom  $i$  from true to false, assuming that atom  $i$  is currently true. It is computed in a way similar to  $g01$ .  $gain[i]$  represents the change in satisfied clauses when changing the value of atom  $i$ . It is implemented using a conditional expression in terms of  $g01[i]$ ,  $g10[i]$ , and the current value of atom  $i$ . No simulation is necessary in the resulting statement.

The GSAT statement can be made even more incremental. Since GSAT only selects the move with the best objective value, it is possible to maintain these candidate moves incrementally. The only change is to add the two invariants

$maxGain$ : **int** =  $\text{max}(i \text{ in } 1..n) gain[i]$ ;  
 $Candidates$ : **{int}** =  
 $\{i : \text{int} \mid \text{select } i \text{ from } 1..n \text{ where } gain[i] = maxGain \text{ and } gain[i] \geq 0\}$ ;

Here  $maxGain$  is simply the maximum of all gains and  $Candidates$  describes the set of candidates for flipping as the set of atoms whose gain is positive and maximal. Once the invariants have been described, the neighborhood is defined by flipping one of the candidates. There is no need to use the keyword **best** or a **noDecrease** acceptance criteria, since they are already enforced by the invariants. The complete statement is depicted in Figure 7. Of course, the same transformation can be performed for the tabu search statement.

```

Solve
Data Type:
  clause = record
    p : {int};
    n : {int};
  end;
Constant:
  m: int = ...;
  n: int = ...;
  cl: array[1..m] of clause = ...;
  po: array[ i in 1..n ] of {int} :=
    { c : int | select c from 1..m where i in cl[c].p };
  no: array[ i in 1..n ] of {int} :=
    { c : int | select c from 1..m where i in cl[c].n };
Variable:
  a: array[1..n] of boolean;
Invariant:
  nbtl: array[ i in 1..n ] of int =
    sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
  g01: array[ i in 1..n ] of int =
    sum(j in po[i]) (nbtl[j] = 0) - sum(j in no[i]) (nbtl[j] = 1);
  g10: array[ i in 1..n ] of int =
    sum(j in no[i]) (nbtl[j] = 0) - sum(j in po[i]) (nbtl[j] = 1);
  gain: array[ i in 1..n ] of int =
    if a[i] then g10[i] else g01[i];
  maxGain : int = max(i in 1..n) gain[i];
  Candidates : {int} =
    { i : int | select i from 1..n where gain[i] = maxGain and gain[i] ≥ 0 };
  nbClauseSat : int = sum(i in 1..m) (nbtl[i] > 0);
Satisfiable:
  nbClauseSat = m;
Neighborhood:
  move a[i] := !a[i]
  where i from Candidates;
Start:
  forall(i in 1..n)
    random(a[i]);
Restart:
  forall(i in 1..n)
    random(a[i]);

```

Figure 7. A more incremental statement of GSAT.

### 3. The Language

As mentioned previously, LOCALIZER is an hybrid language embedding aspects of declarative programming within an imperative language. The main declarative tool is, of course, the concept of invariants which specify expressions whose values must be maintained incrementally. Imperative constructs are mostly used to specify state transformations and the starting and restarting states. This section reviews LOCALIZER in more detail and is essentially organized along the textual ordering of LOCALIZER statements.

#### 3.1. The Type Section

The **Type** section of LOCALIZER is used to define record types. Records within LOCALIZER are identical to records found in conventional imperative languages like Pascal or C. They

aggregate a number of named fields, possibly of different types. For instance, the excerpt

```
Edge = record
  o : int;
  d : int;
end;
```

declares a record type *Edge* consisting of two integers (the origin and destination nodes), while the excerpt

```
clause = record
  pl : {int};
  nl : {int};
end;
```

declares a record type *clause* with the two fields *pl* and *nl* of type “sets of integers”.

### 3.2. Constants

The **Constant** section declares the input data and, possibly, some derived data which are useful in writing the statement. All constants are typed, *read only* (i.e., they cannot be modified by assignments), and are initialized. As shown later on, there are various ways to initialize data in LOCALIZER.

#### 3.2.1. Data Types

##### Basic Data Types

The basic data types supported by LOCALIZER are integers, booleans and floats. The excerpt

```
n : int = 10;
f : float = 3.14;
b : boolean = true;
```

declares an integer *n* whose value is 10, a float whose value is 3.14, and a Boolean. Integers can range from  $-2^{31} + 1$  to  $2^{31} - 1$  and floats are double-precision floating-point numbers.

##### Arrays

LOCALIZER supports multi-dimensional arrays of arbitrary types. The declaration

```
a : array[1..5] of int = [6,1,4,5,7];
```

defines a one-dimensional array  $a$  of integers which is initialized by the vector [6, 1, 4, 5, 7]. The declaration

```
a : array[1..2,1..2] of int = [[1,2],[3,4]];
```

declares a matrix of integers.

### Records

As mentioned previously, records can be used in LOCALIZER to cluster together related data. For instance, the declaration

```
p : array[1..3] of Edge := [< 1, 2 >, < 2, 3 >, < 3, 4 >];
```

defines  $p$  as an array of 3 edges. Each edge is initialized with a tuple. A tuple is compatible with a record type if it has the same number of fields and the type of each field is compatible with the type of the corresponding field.

### Sets

Finally, LOCALIZER supports sets of arbitrary types. For instance, the declaration

```
a : {Edge} = { <1,4>, <2,3>, <3,4>, <2,1>};
```

declares and initializes a set of edges.

### 3.2.2. Inline and Offline Initializations

Constants can be initialized inline as in all previous examples. They can also be initialized *offline* in the **Init** section to separate the model from the instance data, which is usually a good practice. The excerpt

```
f : float = ...;
```

declares a float  $f$  whose initialization is given in the **Init** section. The **Init** section consists of a set of pairs (identifier,value) and, of course, the type of the initialization must match the type of the declaration. Offline initializations can be used for arbitrary types. For instance, the Boolean satisfiability statement may contain an initialization section of the form

```
Init:
  n = 6;
  m = 11;
  cl = [<{1,2,3},{}>,<{4,5,6},{}>,
        <{},{1,2}>,<{},{1,3}>,<{},{2,3}>,
        <{},{4,5}>,<{},{4,6}>,<{},{5,6}>,
        <{},{1,4}>,<{},{2,5}>,<{},{3,6}>];
```

$\langle \text{expr} \rangle$	$::= \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle$ $::= \text{if } \langle \text{expr} \rangle \text{ then } \langle \text{expr} \rangle \text{ else } \langle \text{expr} \rangle$ $::= - \langle \text{expr} \rangle$ $::= \langle \text{aggr} \rangle ( \langle \text{identifier} \rangle \text{ in } \langle \text{Set Body} \rangle ) \langle \text{expr} \rangle$ $::= ( \langle \text{expr} \rangle )$ $::= \text{constant literal}$ $::= \langle \text{Set Expr} \rangle$ $::= \langle \text{expr} \rangle . \langle \text{identifier} \rangle$ $::= \langle \text{expr} \rangle [ \langle \text{expr}_1 \rangle , \dots , \langle \text{expr}_n \rangle ]$ $::= \langle \text{function} \rangle ( \langle \text{expr}_1 \rangle , \dots , \langle \text{expr}_n \rangle )$ $::= \{ \langle \text{Set Body} \rangle \}$ $::= \{ \langle \text{expr} \rangle \{ , \langle \text{expr} \rangle \}$ $::= \{ \langle \text{identifier} \rangle : \langle \text{typed} \rangle \mid \{ \langle \text{select} \rangle \} \}$
$\langle \text{op} \rangle$	$::= [ \text{and} \mid \text{or} \mid < \mid > \mid = \mid \leq \mid \geq \mid <> \mid + \mid - \mid * \mid / \mid \% \mid \uparrow \mid \text{union} ]$
$\langle \text{aggr} \rangle$	$::= [ \text{sum} \mid \text{prod} \mid \text{min} \mid \text{max} \mid \text{argmin} \mid \text{argmax} ]$
$\langle \text{Set Body} \rangle$	$::= \langle \text{expr} \rangle .. \langle \text{expr} \rangle \mid \langle \text{expr} \rangle$ $::= \langle \text{expr} \rangle .. \langle \text{expr} \rangle$ $::= \langle \text{expr} \rangle$
$\langle \text{select} \rangle$	$::= \text{select } \langle \text{identifier} \rangle \text{ from } \langle \text{range} \rangle [ \text{where } \langle \text{expr} \rangle ]$ $::= \text{select } \langle \text{identifier} \rangle \text{ from } \langle \text{expr} \rangle [ \text{where } \langle \text{expr} \rangle ]$
$\langle \text{range} \rangle$	$::= \langle \text{expr} \rangle .. \langle \text{expr} \rangle$

Figure 8. Expression syntax.

Here,  $cl$  is initialized with a vector of 11 tuples. Each tuple is a pair of sets. The first set of each pair is associated with the first field of the record of type *clause* and denotes the set of positive literals in that clause. The second set is matched with the second field of the record and denotes the negative atoms of the clause.

### 3.2.3. Generic Data

LOCALIZER also supports the concepts of generic data which was introduced in NUMERICA [26]. The basic idea here is to initialize the data using an expression which may depend on parameters of the declaration. Genericity is especially attractive to define derived data which are then used to simplify the statement. In the case of the fully incremental version of GSAT (see Figure 7), it is important to know the clauses where an atom  $i$  appears positively (and negatively). This information is derived from the data  $cl$  using the generic declarations

**Constant:**

$po: \text{array}[i \text{ in } 1..n] \text{ of } \{\text{int}\} ::= \{c : \text{int} \mid \text{select } c \text{ from } 1..m \text{ where } i \text{ in } cl[c].p\};$   
 $no: \text{array}[i \text{ in } 1..n] \text{ of } \{\text{int}\} ::= \{c : \text{int} \mid \text{select } c \text{ from } 1..m \text{ where } i \text{ in } cl[c].n\};$

There are a couple of important points to stress here. First, the declarations use parameters which range over the index sets of the array. For instance, parameter  $i$  ranges over  $1..n$ , the set of atoms. Second, these parameters are used in the expression defined on the right-hand side of  $:=$  to specify the value of the array at the given position. In the GSAT example,  $po[i]$  is defined as the set of clauses where atom  $i$  appears positively. The expressions allowed for the right-hand side are very general and their syntax is given in Figure 8. Figure 9 describes the signature of the primitive functions which have the obvious meanings.

Arithmetic	Set related	Output
<code>min2(int, int) → int</code>	<code>size({T}) → int</code>	<code>print(...) → void</code>
<code>max2(int, int) → int</code>	<code>random({T}) → T</code>	<code>println(...) → void</code>
<code>floor(float) → int</code>	<code>minof({T}) → T</code>	
<code>ceil(float) → int</code>	<code>insert({T}, T) → void</code>	
<code>round(float) → int</code>	<code>remove({T}, T) → void</code>	
<code>exp(float) → float</code>		
<code>time() → int</code>		

Figure 9. Primitive functions.

### 3.3. Variables

Variables are of course fundamental in LOCALIZER, since they define the state of the computation. Variables are declared in the same way as constants, except that they are not initialized. They are generally given an initial value in the **Start** section and modified in the **Neighborhood** and **Restart** sections. As should be clear from the examples, LOCALIZER has an assignment operator, whose right-hand side is an expression.

### 3.4. Invariants

Invariants are the key concept provided by LOCALIZER to support the design of local search algorithms. Syntactically, invariants are simply generic data. However, invariants are not static since they may contain variables and/or other invariants. As a consequence, the values of invariants are state-dependent and LOCALIZER is responsible to update them after each state transition. The following excerpt illustrates invariants from a graph-coloring statement where  $C[i]$  refers to another invariant whose type is  $\{int\}$  (set of integers).

<pre> Empty : {int} := { i : int   <b>select</b> i <b>from</b> 1..n <b>where</b> size(C[i])=0}; B : <b>array</b>[k in 1..n] <b>of</b> {Edge} := { &lt; i, j &gt; : Edge   <b>select</b> i <b>from</b> C[k] <b>select</b> j <b>from</b> C[k] <b>where</b> A[i, j] }; </pre>
--

In addition to the standard expressions, invariants can also be defined in terms of **distribute** and **dcount**. They take as input a one-dimensional array of integers  $A$  and two sets  $I$  and  $O$ .  $I$  must be a subset of the index set of  $A$ . The result type of a **distribute** expression is one-dimensional array of set of integers  $B$  whose index set is  $O$ . i.e.,  $B : \mathbf{array}[O] \mathbf{of} \{int\}$ . The result type of a **dcount** expression is one-dimensional array of integers whose index set is  $O$ . Their meanings are given by the following equivalences:

$$\begin{aligned}
B = \mathit{distribute}(A, I, O) &\Leftrightarrow B[i] = \{j \in I \mid A[j] = i\} \forall i \in O \\
B = \mathit{dcount}(A, I, O) &\Leftrightarrow B[i] = \mathbf{size}(\{j \in I \mid A[j] = i\}) \forall i \in O
\end{aligned}$$

```

(statement) ::= (statement) ; (statement)
             ::= (identifier) : (type)
             ::= (lvalue) := (expr)
             ::= if (expr) then (statement) [ else (statement) ] endif
             ::= forall( (identifier) in (Set Body) ) (statement)
             ::= while (expr) do (statement)
             ::= return (expr)
             ::= choose (identifier) from (expr) [ (optClause) ]
(optClause) ::= minimizing (expr)
             ::= maximizing (expr)
             ::= such that (expr) [minimizing (expr) [with (letBlock) ]]
             ::= such that (expr) [maximizing (expr) [with (letBlock) ]]
(letBlock)  ::= { (identifier1) = (expr1) ; ... ; (identifiern) = (exprn) }

```

Figure 10. The syntax of statements.

These expressions were introduced because of their ubiquity in practical applications. It is important to stress that the data declared by invariants cannot appear in left-hand sides of assignment statements.

### 3.5. Operators

The **Operator** section contains function definitions. Functions in LOCALIZER are essentially similar to functions in C and use traditional assignment, conditional, and iterative statements, as well as recursion. In addition, LOCALIZER provides some constructs which are useful for local search algorithms. The syntax of statements in LOCALIZER is sketched in Figure 10. The **forall** instruction provides a convenient way of iterating over the elements of a set. The **choose** instruction can be used to select an element from a set in a (don't care) nondeterministic way. It is possible to filter elements of the given set or to select an element optimizing a function. The following examples illustrate the various forms of choose statements:

```

choose i from A;
choose i from A minimizing D[i];
choose i from A such that gain[i] > 0;
choose i from A such that gain[i] > 0 maximizing D[i];

```

The entire state, including constants, variables, and invariants, are accessible to functions. However, only variables and locals can be modified by assignments. Note that functions and imperative constructs can be used in the **Start** and **Restart** sections, in the specification of the state transformation of the neighborhood, in the actions associated with specific acceptance criteria, and in the **from** instructions that appear in the **where** clause of the **move** instruction. Typical examples of functions were introduced in the statements of Figures 5 and 6.

### 3.6. Neighborhood

The neighborhood is specified in LOCALIZER with a move instruction of the form:

```

move  $op(x_1, \dots, x_n)$ 
where
   $x_1$  from  $S_1$ ;
  ...
   $x_n$  from  $S_n$ 
  [ accept when (AcceptanceCriterion) ]

```

This instruction uses both declarative and procedural components. The first part of the statement specifies, with the imperative code  $op(x_1, \dots, x_n)$ , the transformation of the current state into one of its neighbors. The second part starting with keyword **where** specifies the objects used in the imperative code. The modeling effort is primarily devoted to the definition of the sets  $S_i$  and the invariants they are based on. The syntax of **move** instructions is depicted in Figure 11.

Once an element of the neighborhood has been selected, LOCALIZER determines if it is an appropriate move. The acceptance criterion lists boolean conditions that are to be tried in sequence. As soon as the state satisfies one of the conditions, it is accepted and the state transformation is performed. The criterion is build according to the syntax

```

⟨AcceptanceCriterion⟩ ::= ⟨AcceptanceStatement⟩
                       ::= in resulting state ⟨AcceptanceStatement⟩
                       ::= in current state ⟨AcceptanceStatement⟩

⟨AcceptanceStatement⟩ ::= ⟨AcceptanceCondition⟩
                       ::= ⟨AcceptanceCondition⟩ → ⟨Statement⟩

⟨AcceptanceCondition⟩ ::= always
                       ::= improvement
                       ::= noDecrease
                       ::= ⟨expr⟩
                       ::= ⟨AcceptanceCondition⟩ and ⟨AcceptanceCondition⟩
                       ::= ⟨AcceptanceCondition⟩ or ⟨AcceptanceCondition⟩
                       ::= not ⟨AcceptanceCondition⟩

```

Acceptance criteria are, by default, evaluated in the new state. This new state must be constructed, which induces the update of all invariants. It is also possible to specify that the acceptance criterion should be evaluated in the current state by using the keyword **in current state**. Using the current state to evaluate moves may produce significant improvements in efficiency.

### 3.7. The Objective Function

The objective function is stated in a separate section and is used to assess the “quality” of a given state. The objective function can in fact be viewed as an invariant which is maintained

```

<neighborhood> ::= move op( $x_1, \dots, x_n$ )
                where
                    <letExpr1>
                    ...
                    <letExprn>
                [ accept when <AcceptanceCriterion> ]
<letExpr> ::= <id> from { <Set Body> } [ <optClause> ]
           ::= <id> from { <expr1> , ... , <exprn> } [ <optClause> ]
           ::= <id> from { <id> : <type> | <select1> ... <selectn> } [ <optClause> ]
           ::= <id> from <lvalue> [ <optClause> ]
           ::= <id> = <expr>

```

Figure 11. The syntax of **move** instructions.

by LOCALIZER and used to evaluate moves. Note that the objective function is optional and is only used when the acceptance criteria are evaluated in the resulting state.

### 3.8. Termination Criteria

Termination is handled via several sections and depends on the nature of the problem. Each statement starts with a keyword that is either `solve` or `optimize` to specify either a decision or an optimization problem.

#### *The Satisfiable Section*

The (optional) **Satisfiable** section is used to specify whether a state is a solution. In decision problems, LOCALIZER terminates whenever this is the case. In optimization problems, LOCALIZER updates the best solution whenever the state is a solution which improves the best bound found so far. When the section is omitted, LOCALIZER assumes that every state is a solution.

#### *The Local and Global Conditions*

These sections, which are optional, specify Boolean predicates which are used to control the innermost and outermost loops of the computation model. For instance, a simulated annealing statement can use the local condition to implement a *cutOff* strategy that terminates the innermost loop and drops the temperature whenever the evaluation function has been updated sufficiently many times.

#### *The Parameter Section*

This section, also optional, is used to override the default values of some parameters of the system. Typical uses redefine the `maxSearches` and `maxTrials` parameters of the computation model.

#### 4. Implementation

This section reviews the implementation of invariants which are the cornerstone of LOCALIZER. Informally speaking, invariants are implemented using a planning/execution model. The planning phase generates a specific order for propagating the invariants, while the execution phase actually performs the propagation. This model makes it possible to propagate only differences between two states and mimics, to a certain extent, the way specific local search algorithms are implemented.

The planning/execution model imposes some restrictions on the invariants. These restrictions intuitively, makes sure that there is an order in which the invariants can be propagated so that a pair (variable,invariant) is considered at most once. Various such restrictions can be imposed. Static invariants can be ordered at compile time and are thus especially efficient. However, static invariants rule out some interesting statements for scheduling and resource allocation problems. Dynamic invariants still make it possible to produce an ordering so that a pair (variable,invariant) is considered at most once. However, dynamic invariants require to interleave the planning and execution phases. Dynamic invariants seem to be a good compromise between efficiency and expressiveness and generalize traditional finite differencing approaches.

The rest of this section is organized as follows. The algorithms use normalized invariants and Section 4.1 reviews the normalization process. Section 4.2 describes static invariants and their implementation. This should give readers a preliminary understanding of the implementation. Section 4.3 describes dynamic invariants and their implementation. This section only considers arithmetic invariants but it is not difficult to generalize these results to invariants over sets.

##### 4.1. Normalization

The invariants of LOCALIZER are rewritten into primitive invariants by flattening expressions and arrays. The primitive invariants are of the form:

$$\begin{array}{l} x := c \\ x := y \oplus z \\ x := \prod(x_1, \dots, x_n) \\ x := \text{element}(e, x_1, \dots, x_n) \end{array}$$

where  $c$  is a constant,  $x, y, z, x_1, \dots, x_n$  are variables,  $\oplus$  is an arithmetic operator such as  $+$  and  $*$  or an arithmetic relation such as  $\geq$ ,  $>$  and  $=$ , and  $\prod$  is an aggregate operator such as **sum**, **prod**, **max**, **min**, **argmax**, and **argmin**. Relations return 1 when true and 0 otherwise. An invariant

$$x := \text{element}(e, x_1, \dots, x_n)$$

assigns to  $x$  the element in position  $e$  in the list  $[x_1, \dots, x_n]$ . This last invariant is useful for arrays which are indexed by expressions containing variables.

At any given time, LOCALIZER maintains a set of invariants  $\mathcal{I}$  over variables  $\mathcal{V}$ . Given an invariant  $I \in \mathcal{I}$  of the form  $x := e$  (where  $e$  is an expression),  $\text{def}(I)$  denotes  $x$  while  $\text{exp}(I)$

denotes  $e$ . Given a set of invariants  $\mathcal{I}$  over  $\mathcal{V}$  and  $x \in \mathcal{V}$ ,  $invariants(x, \mathcal{I})$  returns the subset of invariants  $\{I_i\} \subseteq \mathcal{I}$  such that  $x$  occurs in  $exp(I_i)$ . The set of variables  $\mathcal{V} \setminus \{def(I) | I \subseteq \mathcal{I}\}$  are the variables which are the parameters of the system of invariants. These variables only can be modified in the neighborhood definitions. Note also that a variable  $x$  can be defined by at most one invariant, i.e., there exist at most one  $I \subseteq \mathcal{I}$  such that  $def(I) = x$ .

## 4.2. Static Invariants

The basic assumption behind the LOCALIZER implementation is that invariants only change marginally when moving from one state to one of its neighbors. Consequently, the goal of the implementation is to run in time proportional to the amount of changes. More precisely, the implementation makes sure that a pair (*variable, invariant*) is considered at most once, i.e., when the variable is updated, the invariant is updated but it will never be reconsidered because of that variable.

To achieve this goal, the implementation uses a planning/execution model where the planning phase determines an ordering for the updates and the execution phase actually performs them. The existence of a suitable ordering is guaranteed by the restrictions imposed on the invariants by the system. Note also that planning/execution models are often in graphical constraint systems (e.g., [2]).

This section describes static invariants which impose a static restriction. Although this restriction may seem strong, it accommodates many statements for applications such as satisfiability and graph coloring to name a few and is used in all finite differencing systems we are aware of. The main practical limitation is that elements of arrays cannot depend on other elements in the same array. This restriction is lifted by dynamic invariants. Note, however, that static invariants have the nice property that the planning phase can be entirely performed at compile time.

### 4.2.1. The Planning Phase

The basic idea behind static invariants is to require the existence of a topological ordering on the variables. This topological ordering is obtained by associating a topological number  $t(x)$  with each variable  $x$ . The topological number of an invariant  $I$  is simply  $t(def(I))$ . The topological numbers are obtained from constraints derived from the invariants.

*Definition.* The topological constraints of invariant  $I$ , denoted by  $tc(I)$ , are defined as follows:

$$\begin{aligned} tc(x := c) &= \{t(x) = 0\} \\ tc(x := y \oplus z) &= \{t(x) = \max(t(y), t(z)) + 1\} \\ tc(x := \prod(x_1, \dots, x_n)) &= \{t(x) = \max(t(x_1), \dots, t(x_n)) + 1\} \\ tc(x := element(e, x_1, \dots, x_n)) &= \{t(x) = \max(t(e), t(x_1), \dots, t(x_n)) + 1\} \end{aligned}$$

*Definition.* The topological constraints of a set of invariants  $\mathcal{I}$ , denoted by  $tc(\mathcal{I})$ , is  $\bigcup_{I \in \mathcal{I}} tc(I)$ .

```

procedure execute( $\mathcal{I}, \mathcal{M}, t$ )
begin
   $Q := \{\langle x, I \rangle \mid x \in \mathcal{M} \wedge I \in \mathcal{I} \setminus \{\exists, \forall\} \cup \{\exists, \forall\}(\mathcal{I})\}$ 
  while  $Q \neq \emptyset$  do
     $\langle x, I \rangle := \text{POP}(Q, t)$ ;
    propagate( $x, I, \mathcal{I}, Q$ );
     $Q := Q \cup Q' \setminus \{\langle x, I \rangle\}$ ;
  endwhile
end

function POP( $Q, t$ )
  pre:  $Q$  is not empty
  post:  $\langle x, I \rangle \in Q$  such that  $\forall \langle x', I' \rangle \in Q : t(I') \geq t(I)$ 

```

Figure 12. The execution phase for static invariants.

*Definition.* A set of invariants  $\mathcal{I}$  over  $t$  is static if there exists an assignment  $t: \mathcal{V} \rightarrow \mathcal{N}$  such that  $t$  satisfies  $tc(\mathcal{I})$ .

The planning phase for static invariants consists of finding the topological assignment. The planning phase can be performed at compile time since the topological constraints do not depend on the values of the variables in a given state.

#### 4.2.2. The Execution Phase

The execution phase is given a set of variables  $\mathcal{M}$  which have been updated and a topological assignment  $t$ . It then propagates the changes according to the topological ordering. The algorithm uses a queue which contains pairs of the form  $\langle x, I \rangle$ . Intuitively, such a pair means that invariant  $I$  must be reconsidered because variable  $x$  has been updated. The main step of the algorithm consists of popping the pair  $\langle x, I \rangle$  with the smallest  $t(I)$  and to propagate the change, possibly adding new elements to the queue. The algorithm is shown in Figure 12.

#### 4.2.3. Propagating the Invariants

To complete the description of the implementation of static invariants, it remains to describe how to propagate the invariants themselves. The basic idea here is to associate two values  $x^o$  and  $x^c$  with each variable  $x$ . The value  $x^o$  represents the value of variable  $x$  at the beginning of the execution phase, while the value  $x^c$  represents the current value of  $x$ . At the beginning of the execution phase,  $x^o = x^c$  of course. By keeping these two values, it is possible to compute how much a variable has changed and to update the invariants accordingly. For instance, the propagation of the invariant

$$x := \text{sum}(x_1, \dots, x_n)$$

Table 1. Space and time complexity bounds for static invariants.

Invariant	Time	Space
$x := c$	$\Theta(1)$	$\Theta(1)$
$x := y \oplus z$	$\Theta(1)$	$\Theta(1)$
$x := \min(i \text{ in } \{1..n\})x_i$	$O(\log n)$	$\Theta(n)$
$x := \max(i \text{ in } \{1..n\})x_i$	$O(\log n)$	$\Theta(n)$
$x := \operatorname{argmin}(i \text{ in } \{1..n\})x_i$	$O(\log n)$	$\Theta(n)$
$x := \operatorname{argmax}(i \text{ in } \{1..n\})x_i$	$O(\log n)$	$\Theta(n)$
$x := \operatorname{sum}(i \text{ in } \{1..n\})x_i$	$\Theta(1)$	$\Theta(n)$
$x := \operatorname{prod}(i \text{ in } \{1..n\})x_i$	$\Theta(1)$	$\Theta(n)$

is performed by a procedure

```

procedure propagate( $x_i, x := \operatorname{sum}(x_1, \dots, x_n), \mathcal{I}, Q$ )
begin
   $x^c := x^c + (x_i^c - x_i^o)$ ;
  if  $x^c \neq x^o$  then
     $Q := \operatorname{invariants}(\mathcal{I}, x)$ ;
end

```

The procedure updates  $x^c$  according to the change of  $x_i$ . Note that, because of the topological ordering,  $x_i^c$  has reached its final value. Note also that  $x^c$  is not necessarily final after this update, because other pairs  $\langle x_j, x := \operatorname{sum}(x_1, \dots, x_n) \rangle$  may need to be propagated. Finally, note that the static elementary invariants described here obey the complexity bounds reported in Table 1.

### 4.3. Dynamic Invariants

Static invariants are attractive since the planning phase can be performed entirely at compile time. However, there are interesting applications in the areas of scheduling and resource allocation where sets of invariants are not static. This section introduces dynamic invariants to broaden the class of invariants accepted by LOCALIZER. Dynamic invariants are updated by a series of planning/execution phases where the planning phase takes place at execution time.

#### 4.3.1. Motivation

The main restriction of static invariants comes from the invariant

$$x := \operatorname{element}(e, y_1, \dots, y_n)$$

The static topological constraint for this invariant is

$$t(x) = \max(t(e), t(y_1), \dots, t(y_n)) + 1$$

and it prevents LOCALIZER from accepting expressions where some elements of an array may depend on some other elements of the same array. This constraint is strong, because the value of  $e$  is not known at compile time. In fact, it may not even be known before the start of the execution phase since some invariants may update it.

However, there are many applications in scheduling or resource allocation where such invariants occur naturally. For instance, a scheduling application may be modeled in terms of an invariant

$$start[3] := \max(end[prec[3]], end[disj[3]]);$$

where  $start[i]$  represents the starting date of task  $i$ ,  $prec[i]$  the predecessor of the task in the job and  $disj[i]$ , the predecessor of task  $i$  in the disjunction. Variable  $disj[i]$  is typically updated during the local search and the above invariant is normalized into a set of the form:

$$\begin{aligned} start_3 &:= \max(p, d) \\ p &:= \text{element}(prec_3, end_1, \dots, end_n) \\ d &:= \text{element}(disj_3, end_1, \dots, end_n) \end{aligned}$$

Of course, such an application has also invariants of the form

$$\begin{aligned} end_1 &:= start_1 + duration_1 \\ &\vdots \\ end_3 &:= start_3 + duration_3 \\ &\vdots \\ end_n &:= start_n + duration_n \end{aligned}$$

implying that the resulting set of invariants is not static.

#### 4.3.2. Overview of the Approach

The basic idea behind dynamic invariants is to evaluate the invariants by levels. Each invariant is associated with one level and, inside one level, the invariants are static. Once a level is completed, planning of the next level can take place using the values of the previous levels since lower levels are never reconsidered. With this computation model in mind, the topological constraint associated with an invariant

$$x := \text{element}(e, y_1, \dots, y_n)$$

can be reconsidered. The basic idea is to require that  $e$  be evaluated before  $x$  (i.e., the level of  $x$  is the level of  $e + 1$ ). Once  $e$  is updated, then it is easy to find a weaker topological constraint since the value of  $e$  is known. The invariant can be simplified to

$$x := y_{e^c}$$

and  $x$  only depends on one element in  $\{y_1, \dots, y_n\}$ . The planning phase is thus divided in two steps. A first step, which can be carried out at compile time, partitions the invariants in levels. The second step, which is executed at runtime, topologically sorts the invariants within each level whenever the invariants at the lower level have been propagated.

### 4.3.3. Formalization

The basic intuition is formalized in terms of two assignments  $l: \mathcal{V} \rightarrow \mathcal{N}$  and  $t: \mathcal{V} \rightarrow \mathcal{N}$  and of two sets of constraints.

*Definition.* The level constraints associated with an invariant  $I$ , denoted by  $lc(I)$ , are defined as follows:

$$\begin{aligned} lc(x := c) &= \{l(x) = 0\} \\ lc(x := y \oplus z) &= \{l(x) = \max(l(y), l(z))\} \\ lc(x := \prod(x_1, \dots, x_n)) &= \{l(x) = \max(l(x_1), \dots, l(x_n))\} \\ lc(x := \mathit{element}(e, x_1, \dots, x_n)) &= \{l(x) = \max(l(e) + 1, l(x_1), \dots, l(x_n))\} \end{aligned}$$

The level constraints are not strong except for the invariant **element** where the level of  $x$  is strictly greater than the level of  $e$ . Informally, it means that  $e$  must be evaluated in an earlier phase than  $x$ .

*Definition.* The level constraints associated with a set of invariants  $\mathcal{I}$ , denoted by  $l(\mathcal{I})$ , are simply  $\bigcup_{I \in \mathcal{I}} lc(I)$ .

*Definition.* A set of invariants  $\mathcal{I}$  is serializable if there exists an assignment  $l: \mathcal{V} \rightarrow \mathcal{N}$  satisfying  $lc(\mathcal{I})$ .

A serializable set of invariants can be partitioned into a sequence  $\langle \mathcal{I}_0, \dots, \mathcal{I}_p \rangle$  such the invariants in  $\mathcal{I}_i$  have level  $i$ . This serialization can be performed at compile-time. The second step consists of ordering the invariants inside each partition. This ordering can only take place at runtime, since it is necessary to know the values of some invariants to simplify the **element** invariants.

*Definition.* Let  $S$  be a computation state and let  $S(x)$  denote the value of  $x$  in  $S$ . The topological constraints associated with an invariant  $I$  wrt  $S$ , denoted by  $tc(I, S)$ , are defined as follows:

$$\begin{aligned} tc(x := c, S) &= \{t(x) = 0\} \\ tc(x := y \oplus z, S) &= \{t(x) = \max(t(y), t(z)) + 1\} \\ tc(x := \prod(x_1, \dots, x_n), S) &= \{t(x) = \max(t(x_1), \dots, t(x_n)) + 1\} \\ tc(x := \mathit{element}(e, x_1, \dots, x_n), S) &= \{t(x) = t(x_{S(e)}) + 1\} \end{aligned}$$

*Definition.* The topological constraints associated with a set of invariants  $\mathcal{I}$  wrt a state  $S$ , denoted by  $tc(\mathcal{I}, S)$ , are simply  $\bigcup_{I \in \mathcal{I}} tc(I, S)$ .

*Definition.* A set of invariants  $\mathcal{I}$  is static wrt a state  $S$  if there exists an assignment  $t: \mathcal{V} \rightarrow \mathcal{N}$  satisfying  $tc(\mathcal{I}, S)$ .

The main novelty of course is in the invariant **element** where the topological constraint can ignore  $e$  since its value is known. In addition, since the final value of  $e$  is known, the topological constraints can be made precise since the element  $y_{e^c}$  that  $x$  depends upon is known.

```

procedure execute( $\mathcal{I}, \mathcal{M}$ )
begin
   $\langle \mathcal{I}_1, \dots, \mathcal{I}_p \rangle := \text{serialize}(\mathcal{I});$ 
  for(  $i := 0; i \leq p; i++$ ) do
     $t := \text{plan}(\mathcal{I}_i);$ 
    execute( $\mathcal{I}_i, \mathcal{M}, t$ );
  endfor
end

```

Figure 13. The execution algorithm for dynamic invariants.

*Definition.* Let  $S_0$  be a computation state. A set of invariants  $\mathcal{I}$  is dynamic wrt  $S_0$  if

1.  $\mathcal{I}$  is serializable and can be partitioned into a sequence  $\langle \mathcal{I}_0, \dots, \mathcal{I}_p \rangle$ ;
2.  $\mathcal{I}_i$  is static wrt  $S_i$  where  $S_i$  ( $i > 0$ ) is the state obtained by propagating the invariants  $\mathcal{I}_{i-1}$  in  $S_{i-1}$ .

Of course, dynamic invariants cannot be recognized at compile-time and may produce an execution error at runtime when LOCALIZER is planning a level. Note that the propagation of a dynamic invariant like `element` requires  $\Theta(1)$  in time and space.

#### 4.3.4. The Execution Algorithm

The new execution algorithm is a simple generalization of the static algorithm and is shown in Figure 13. Note the planning step which is called for each level.

#### 4.3.5. Performance Considerations

The previous section described, to some extent, the space and time complexities of the various elementary invariants. However, this is not sufficient to analyze a LOCALIZER statement. It is also important to consider elementary invariants globally, since it determines the number of  $\langle x, I \rangle$  pairs that must be reconsidered. It is thus not surprising that different models can lead to substantial differences in efficiency. For instance, consider the incremental GSAT statement of Figure 7. The cost for updating the `nbtl` array when atom  $k$  is flipped is linear in the number of clauses where atom  $k$  appears. The propagation of the `nbtl` invariants is linear (thus optimal) in the size of the changes in input and output. Similarly, if  $m$  is the total number of atoms appearing in the affected clauses, then updating `g01` and `g10` cost  $O(m)$ . Once again, the propagation of these two invariants is optimal with respect to the amount of change in input and output. Any statement can be analyzed in a similar fashion and interested readers could consult [12] for more information.

Table 2. GSAT: Experimental results.

	<i>V</i>	<i>C</i>	<i>I</i>	<i>L</i>	<i>G</i>	<i>R</i>
1	100	430	500	19.54	6.00	3.26
2	120	516	600	40.73	14.00	2.91
3	140	602	700	54.64	14.00	3.90
4	150	645	1500	154.68	45.00	3.44
5	200	860	2000	873.11	168.00	5.20
6	250	1062	2500	823.06	246.00	3.35
7	300	1275	6000	1173.78	720.00	1.63

## 5. Experimental results

This section summarizes some preliminary results on the implementation of LOCALIZER (about 50,000 lines of C++). The goal is not to report the final word on the implementation but rather to suggest that LOCALIZER can be implemented with an efficiency comparable to specific local search algorithms. To demonstrate practicability, we experimented with LOCALIZER on several problems: GSAT, graph coloring, graph partitioning, and job-shop scheduling.

### 5.1. GSAT

GSAT is generally recognized as a fast and very well implemented system. The experimental results were carried out as specified in [22]. Table 2 gives the number of variables (*V*), the number of clauses (*C*) and **MaxTrials** (*I*) for each size of benchmarks, as well as the CPU times in seconds of LOCALIZER (*L*), the CPU times in seconds of GSAT (*G*) as reported in [22], and the ratio *L/G*. The times of GSAT are given on a SGI Challenge with a 70 MHz MIPS R4400 processor. The times of LOCALIZER were obtained on a SUN SPARC-10 40MHz and scaled by a factor 1.5 to account for the speed difference between the two machines. LOCALIZER times are for the incremental statement presented in Section 2. Note that this comparison is not perfect (e.g., the randomization may be different) but it is sufficient for showing that LOCALIZER can be implemented efficiently.

The class of formula used for the experiment correspond to hard instances where the ratio of  $\frac{nbClause}{nbVars}$  is about 4.3. It is well known that whenever the actual ratio becomes much smaller or much larger, the instance becomes extremely easy to solve. For formulas with less than 200 variables, 100 benchmarks were generated. For larger formulas, 10 benchmarks were generated. The results reported in the table aggregates over 10 runs for each formula for a total of 1000 runs (100 for large formulas). As can be seen, the distribution of ratios *L/G* is uniform over the range of benchmarks. This indicates that the LOCALIZER statement scales in the same way than GSAT. The gap between the two systems is about one machine generation (i.e., on modern workstations, LOCALIZER runs as efficiently as GSAT on machines of three years ago), which is really acceptable given the preliminary nature of our (unoptimized) implementation.

## 5.2. Graph Coloring

Graph coloring was the object of an extensive experimental evaluation in [11] and this section reports on experimental results along the same lines. The experiments were conducted on graphs of densities 10, 50, and 90 and of sizes 125, 250, and 500. They were also conducted on so-called “cooked” graphs. Because of the nature of the experimental results reported in [11], it is not easy to compare the efficiency of LOCALIZER to the efficiency of their algorithm. As a consequence, a very efficient C implementation of their algorithm was built from scratch by a graduate student who was closely supervised to obtain a very efficient incremental algorithm. As far as we can judge, the timings and the quality of this algorithm seem consistent with those in [11]. In addition, this algorithm is the most efficient implementation built by a graduate student in the combinatorial optimization class at Brown (CS-258) in the last three years (for the given model of course). In the following, we discuss the development time of the two implementations, the quality of the solutions obtained (to make sure that the algorithms are comparable in quality), and the efficiency.

### *Development Time*

The C implementation of the algorithm is about 1500 lines long and required a full week. The LOCALIZER statement is about one page long.

### *Quality of the Solutions*

Table 3 describes the quality of the coloring found by LOCALIZER. These results agree with those of the C implementation and with those reported in [11]. Each row corresponds to a class of graphs and to 100 executions of LOCALIZER on graphs from this class. It describes the problem class (random or cooked graph), the number of vertices ( $V$ ), the edge density ( $D$ ), the size factor parameter ( $S$ ) and report on the various colorings found by LOCALIZER on these graphs and their frequencies. Colorings preceded or succeeded by a dash indicates that the frequency is associated with an open range of colorings. For instance, the first row reports that, on graphs of 125 vertices and density of 50%, 92% of the executions led to a coloring with 19 colors and 8% of the executions led to a coloring with 20 colors. The results are given both for random and cooked graphs and the frequencies are similar for both LOCALIZER and the C implementation.

### *Efficiency*

Table 4 compares the efficiency of LOCALIZER with the C implementation on the same problems. Each row reports the average time of the two implementations for the 100 graphs in each class and computes the slowdown of LOCALIZER. The experiments were performed on a SUN Sparc Ultra-1 running Solaris 5.5.1 and the standard C++ compiler. The average slowdown is 4.82, while minimum and maximum slowdowns are respectively 3.56 and 5.54.

Table 3. Graph coloring: quality of the solutions.

	Data Set			Color Ranges				Frequencies							
	V	D	S					LOCALIZER				C			
r	125	50	3	19	20			92	8			87	13		
r	250	50	4	-32	33	34	35-	9	43	44	4	6	41	50	3
r	500	50	4	55	56	57	58-	4	54	41	1	8	48	36	8
r	125	10	1	6	7	8	-	48	52	-	-	43	54	3	-
r	250	10	1	9	10	11	-	27	70	3	-	29	70	1	-
r	500	10	2	15	16	17	-	1	79	20	-	3	86	11	-
r	125	90	1	-44	45	46	47-	7	30	30	33	15	32	26	27
r	250	90	1	-78	79	80	81-	4	15	35	46	4	19	25	52
r	500	90	1	-143	144	145	146-	30	22	16	32	15	11	38	36
c	125		4	9	-	-	-	100	-	-	-	100	-	-	-
c	250		1	15	-	-	-	100	-	-	-	100	-	-	-
c	500		2	25	26-			71	29			65	35		

Table 4. Graph coloring: efficiency of LOCALIZER.

	Vertices	Density	S	L	C	L/C
r	125	50	3	78.3	18.9	4.50
r	250	50	4	82.8	18.4	4.50
r	500	50	4	633.7	123.4	5.10
r	125	10	1	22.5	4.8	4.60
r	250	10	1	109.2	22.02	4.96
r	500	10	2	159.8	28.8	5.50
r	125	90	1	16.18	4.53	3.56
r	250	90	1	49.32	8.89	5.54
r	500	90	1	162.7	29.6	4.88
c	125		4	22.09	4.18	5.28
c	250		1	37.22	7.79	4.77
c	500		2	240.3	49.9	4.80

On these problems, the average slowdown is slightly higher than a machine generation but it remains reasonable given the preliminary nature of the implementation. This slowdown should also be contrasted with the substantial reduction in development time.

### 5.3. Graph Partitioning

The graph-partitioning problem consists of finding a partition of the vertices of a graph into two sets of equal size which minimizes the number of edges connecting the two sets. The local search algorithm considered here is based on a simulated annealing algorithm presented in [10]. This algorithm relaxes the idea of maintaining a feasible solution and a move consists of selecting a vertex and moving it to the other set. The objective function combines the objective of minimizing the connections between the two sets with the desire to favor balanced solutions. It is given as

$$SB + \alpha * IMB^2$$

Table 5. Graph partitioning: experimental results.

Graph		Results ( $T = 10$ , $TF = 0.95$ , $S = 16$ , $chPerc = 2\%$ , $Cutoff = 10\%$ )							
Class	$V$	$D$	Cut Sizes			Frequencies			LOCAL.
r	124	2	11–13	14–16	17–19	33	38	29	2.22
r		4	55–59	60–65	66–77	30	34	36	2.40
r		8	159–174	175–190	191–	23	53	24	3.24
r		16	481–560	561–640	641–	36	45	19	4.05
r	250	1	20–24	25–29	30–35	19	22	59	4.68
r		2	92–106	107–121	122–131	12	42	36	5.10
r		4	324–343	344–363	364–380	38	43	19	6.50
r		8	828–877	878–927	928–	37	40	23	9.98
r	500	0.5	48–54	55–59	60–66	15	43	42	10.08
r		1	219–231	232–244	245–256	17	52	31	10.76
r		2	637–661	662–686	686–718	10	15	75	14.44
r		4	1661–1701	1702–1741	1742–1824	22	58	20	20.67
r	1000	0.25	90–103	104–118	119–126	41	52	7	19.99
r		0.5	439–455	456–475	476–503	36	43	21	22.62
r		1	1326–1357	1358–1397	1398–1427	33	51	16	29.97
r		2	3253–3319	3320–3394	3394–3466	11	37	52	40.10
Class	$V$	$n\pi d^2$	Cut Sizes			Frequencies			LOCAL.
geom.	500	5	4–13	14–23	24–37	7	58	35	8.26
geom.		10	35–59	60–84	85–123	19	42	39	9.50
geom.		20	148–246	247–346	347–450	41	44	15	11.40
geom.		40	441–840	841–1240	1241–3400	47	20	33	14.50
geom.	1000	5	24–43	44–63	64–78	37	57	6	18.70
geom.		10	65–114	115–164	165–205	16	54	30	21.20
geom.		20	196–399	400–599	600–816	32	60	8	23.84
geom.		40	537–1099	1100–1599	1600–5829	28	49	23	28.56

where  $SB$  is the number of connections,  $IMB$  is the imbalance between the two sets, and  $alpha$  is a parameter of the algorithm.

The problem has been studied experimentally in [10] and, once again, the experiments reported here are based on a similar setting. Table 5 depicts the experimental results of LOCALIZER. The first row gives the setting of our parameters:  $T$  is the starting temperature,  $TF$  is the percentage of reduction of the temperature,  $S$  is the size factor, and the remaining two parameters were described previously. As before, each row describes a problem instance in terms of its class, the number of vertices ( $V$ ), and the edge density ( $D$ ) and reports the number of times LOCALIZER produced a partition with a cut size in the given range. The lower bound of the first range indicates the best solution ever found by LOCALIZER. The last column also reports the average running time.

Table 6 compares LOCALIZER with the results reported in [10]. It is important to mention that [10] explicitly writes that their results are very hard to reproduce, since they chose the temperature of the annealing algorithm according to some preliminary observation of its behavior on each class of graphs. LOCALIZER, in contrast, is always run with the given parameters. In addition, the relevant results are only given for the random graphs in [10].

Table 6. Graph partitioning: comparison results.

Graph		Results (T = 10, TF = 0.95, S = 16, chPerc = 2%, Cutoff = 10%)					
Class	V	D	L.Best	L.Time	J.Best	J.Time	Ratio
random	124	2	11	2.22	13	85.4	38.5
		4	55	2.4	63	82.2	34.3
		8	159	3.24	178	78.1	24.1
		16	481	4.05	449	104.8	25.9
random	250	1	20	4.68	29	190.6	40.7
		2	92	5.1	114	163.7	32.1
		4	324	6.5	357	186.8	28.7
		8	828	9.98	828	223.3	22.4
random	500	0.5	47	10.08	52	379.8	37.7
		1	219	10.76	219	308.9	28.7
		2	635	14.44	628	341.5	23.6
		4	1661	20.67	1744	432.9	20.9
random	1000	0.25	90	19.99	102	729.9	36.5
		0.5	439	22.62	451	661.2	29.2
		1	1326	29.97	1367	734.5	24.5
		2	3253	40.01	3389	853.7	21.3

The times for LOCALIZER are given on a SUN Sparc Ultra-1 running Solaris 5.5.1 and the standard C++ compiler, while the results in [10] are given for a slow VAX-750. In general, the quality of the results produced by LOCALIZER is slightly better than the quality in [10]. Once again, the performance results indicate that LOCALIZER behaves well on these problems.

#### 5.4. Job-Shop Scheduling

To conclude, we report some preliminary results on job-shop scheduling. LOCALIZER has been evaluated on a set of 28 classic benchmarks. The tabu-search statement used for these experiments implements the neighborhood, commonly referred to as  $N1$ , which considers the reversal of exactly one edge on the critical path.  $N1$  has a number of nice properties: It preserves the satisfiability of the solution and the transition graph induced by the neighborhood is such that the optimal solution is reachable from all nodes of the graph. The experiments were conducted in a fashion similar to what is reported in [5]. The parameters were defined as follows:

- The maximal number of searches (maxSearches) is 1
- The maximal number of iterations for the inner loop (maxTrials) is 12000.
- The tabu list has a varying length constrained in between 5 and 30. Moreover, the length is varied according to the rule of [5].
- Contrary to [5], the statement does not use a restarting strategy.

Table 7 reports the preliminary results. These results cannot be really compared with the results of [5], since the neighborhood used in their experiment is  $(RN1 \cup RN2)$  while the statement used here relies on  $N1$  alone. Each row describes a benchmark with its name, the number of jobs and machines and the best solution known (italics denote lower bounds). The quality of the solutions is reported with a coarse histogram. For instance, for *LA16 LOCALIZER* found one makespan in the range [947, 947], 11 in [948, 961], 21 in [962, 975] and 67 in [976, 988]. The times reported in column *LOC* correspond to the algorithm termination (once the 12000 iterations are spent), while column *LO* reports the times required to produce the optimal solution for the first time (this is the time measure used in [5]). The column *Avg.* reports the average running time and *D* reports the distance to the known optimum (or best known lower bound as of 1993) as a percentage. Note also that the value of the optimal solution was not used as a stopping criterion. If this condition was to be used, the running time would vary a lot more. For instance, *LA01* usually produces the optimum solution in about 0.5 second. Interestingly enough, even the simple neighborhood  $N1$  does quite well and finds the optimum solution for 19 benchmarks (out of 38 benchmarks). These frequencies were obtained based on a series of 100 experiments for each benchmark.

In summary, the results seem to indicate that *LOCALIZER* will also compare well on scheduling applications.

## 6. Related Work

This section reviews related research in constraint programming, modeling languages, programming languages and incremental algorithms. It also discusses directions for future work when appropriate.

### 6.1. Constraint Programming Languages

Many constraints programming languages have been defined in the last 15 years to support the solving of combinatorial optimization problems. Well-known representatives include *CHIP*, *Ilog Solver*, *CLP(R)*, *PROLOG IV* and *OZ* and they are based on various programming paradigms (e.g. logicprogramming, functional programming, object-oriented programming). Almost all these languages support the global search paradigm (e.g. branch and bound or constraint satisfaction). More precisely, a program in these languages can be seen as a high-level specification of a global search algorithms. These specifications generally consists of two parts: a declarative component that specifies the set of constraints to be satisfied and an algorithmic component that searches for solutions.<sup>2</sup> These constraint programming languages generally use “logical variables”, i.e., variables that can be assigned once, and the purpose of a constraint program is generally to find values for the variables. Finally, at the core of these languages lie constraint solving algorithms which may be rather sophisticated and include linear and non-linear programming algorithms as well as constraint satisfaction algorithms.

Table 7. Job-shop scheduling: experimental results

Benchmark			Results (MD = 12000, MS = 1, Neighborhood = N1)					
<i>Name</i>	<i>J/M</i>	<i>Opt</i>	<i>Ranges</i>	<i>Freq.</i>	LOC	LO	<i>Avg.</i>	<i>D</i>
LA06	15/5	926	926/926/926/926	100/0/0/0	22.1	0.8	926.0	0
LA07	15/5	890	890/890/890/890	100/0/0/0	24.5	3.1	890.0	0
LA08	15/5	863	863/863/863/863	100/0/0/0	23.7	1.5	863.0	0
LA09	15/5	951	951/951/951/951	100/0/0/0	22.7	1.2	951.0	0
LA10	15/5	958	958/958/958/958	100/0/0/0	21.9	1.1	958.0	0
LA11	20/5	1222	1222/1222/1222/1222	100/0/0/0	25.2	3.6	1222.0	0
LA12	20/5	1039	1039/1039/1039/1039	100/0/0/0	24.0	3.1	1039.0	0
LA13	20/5	1150	1150/1150/1150/1150	100/0/0/0	24.0	6.8	1150.0	0
LA14	20/5	1292	1292/1292/1292/1292	100/0/0/0	22.8	2.4	1292.0	0
LA15	20/5	1207	1207/1207/1207/1207	100/0/0/0	27.4	5.5	1207.0	0
LA16	10/10	945	947/961/975/988	1/11/1/7	35.4	35.2	975.1	3
LA17	10/10	784	784/790/796/801	17/74/8/1	36.1	34.2	786.4	0
LA18	10/10	848	848/857/866/873	1/31/56/12	36.4	35.9	860.1	1
LA19	10/10	842	843/850/857/864	1/25/49/25	37.0	36.5	853.9	1
LA20	10/10	902	902/908/914/918	13/24/59/4	36.7	33.1	909.5	1
LA21	15/10	1048	1060/1078/1096/1114	1/27/57/15	48.4	49.0	1084.9	3
LA22	15/10	927	935/948/961/974	1/28/56/15	47.2	47.9	952.9	3
LA23	15/10	1032	1032/1033/1034/1034	99/0/1/0	49.2	22.0	1032.0	0
LA24	15/10	935	945/959/973/985	2/22/67/9	47.2	47.8	964.3	3
LA25	15/10	977	989/1010/1031/1051	1/39/51/9	46.0	46.7	1015.1	4
ABZ5	10/10	1234	1236/1246/1256/1264	3/34/54/9	36.8	38.4	1248.8	1
ABZ6	10/10	943	943/948/953/958	13/69/16/2	37.4	37.4	946.9	0
ABZ7	20/15	667	686/723/760/797	1/58/39/2	79.3	84.7	721.5	8
ABZ8	20/15	678	688/724/760/796	1/6/70/23	83.3	80.7	747.7	10
ABZ9	20/15	692	715/735/755/774	2/50/43/5	93.9	88.9	735.2	6
MT6	6/6	55	55/55/55/55	100/0/0/0	13.2	1.0	55.0	0
MT10	10/10	930	941/959/977/941	1/35/43/0	36.6	23.6	966.1	4
MT20	20/5	1165	1173/1194/1215/1173	8/67/23/0	29.8	18.7	1186.1	2
ORB1	10/10	1059	1073/1095/1117/1160	1/2/25/72	36.2	36.4	1124.6	6
ORB2	10/10	888	889/896/903/917	3/33/42/22	36.1	36.3	899.6	1
ORB3	10/10	1005	1021/1081/1141/1261	2/89/8/1	37.4	37.6	1060.5	6
ORB4	10/10	1005	1019/1031/1043/1064	1/30/44/25	33.8	34.1	1037.3	3
ORB5	10/10	887	899/911/923/947	1/29/40/30	37.5	37.7	918.7	4
ORB6	10/10	1010	1022/1034/1046/1069	2/26/42/30	35.8	36.1	1041.8	3
ORB7	10/10	397	397/403/409/420	1/5/47/47	38.5	38.3	409.5	3
ORB8	10/10	899	914/932/950/986	1/9/54/36	37.5	37.8	947.8	5
ORB9	10/10	934	934/945/956/976	1/3/35/61	32.4	32.7	959.8	3
OR10	10/10	944	944/956/968/992	2/24/48/26	37.0	36.6	963.5	2

LOCALIZER differs fundamentally from these languages in that it supports the local search paradigm. LOCALIZER is the first language to support the local search paradigm. There have been attempts (e.g., [23]) to embed local search algorithms into constraint languages but LOCALIZER is the first language that supports users in defining their own local search algorithms. A statement in LOCALIZER is a high-level specification of a local search algorithm and LOCALIZER also has declarative and algorithmic components. The declarative component specifies the invariants that must be maintained, while the algorithmic com-

ponent modifies the value of some of the variables to move from neighbors to neighbors. Note that variables in LOCALIZER are closer to variables in imperative languages and that the core of a LOCALIZER statement is a collection of incremental algorithms to maintain invariants. These differences with traditional constraint programming languages are in no way artificial but reflect to the fundamentally different way of organizing global and local search algorithms.

Note finally that CLAIRE [3] is a language that provides tools for implementing global search algorithms (e.g. deduction rules and non-determinism). However, since CLAIRE is based on traditional variables, it can also be used to implement local search algorithms by enhancing its functionality with invariants.

## 6.2. Modeling Languages

Modeling languages such as AMPL [7], GAMS [1], LINDO [21] and NUMERICA [26] have been proposed to simplify the design of mathematical programming problems. These modeling languages focus on stating the problem constraints and support high-level algebraic and set notations to express these constraints from the data. They are mostly independent from the underlying solving algorithms (e.g., linear programming solvers), although they may give access to some of the solver's parameters and options. These modeling languages do not support the search component of traditional constraint programming languages but they have rich data modeling features that make them accessible to a wide audience. An exception is the optimization programming language OPL [25], a recent addition in the field of modeling languages. OPL in fact allows both the specification of constraints as in traditional modeling languages and the definition of search procedures.

LOCALIZER shares with these languages the idea of supporting high-level algebraic and set notations to simplify problem statements. However, there is a fundamental difference between LOCALIZER and these languages: a LOCALIZER statement describes a local search algorithm, not a set of constraints or a global search algorithm.

It is also useful to point out that the scripting languages provided by some modeling languages such as AMPL aims at specifying algorithms in terms of models. Their contributions take place at another level and are orthogonal to LOCALIZER.

## 6.3. Graphical Constraint Systems

Historically, constraints were first used in graphical systems such as Sketchpad [24] and Thinglab [2]. Many of the available constraint-based graphical packages containing constraints in fact support one-way constraints that can be viewed as invariants. LOCALIZER was in fact inspired by these systems and originated in our desire to determine if this approach could provide adequate support for local search algorithms. LOCALIZER, of course, goes far beyond typical one-way constraints found in graphical systems, both in terms of the expressiveness of its invariants and in terms of its implementation technology. LOCALIZER invariants may contain arrays (indexed by variables), sets, graphs and other higher level data

structures. The basic planning/execution model also had to be generalized to interleave the planning and the execution phases to support dynamic invariants.

Much recent work in constraint-based graphics has been concerned with multi-way constraints and constraint hierarchies. In graphical systems, one is often interested in specifying relationships between graphical objects and one-way constraints are not always the best vehicle to support these relations. Multiway constraints specify how to maintain a relationship and constraint hierarchies are used to decide how to choose between different solutions. These functionalities are not needed in LOCALIZER, since invariants are used to build data structures incrementally, not to maintain relationships incrementally.

#### 6.4. Finite Differencing

Finite differencing [16] is a technique used in optimizing compilers to improve efficiency. To review finite differencing, it is useful to start with a simple example. Consider the program

```

for (i:=0;i < n;i++) {
    f := f + i * c;
}
```

The time-consuming part in this program is the multiplication  $i * c$ . The basic idea behind finite differencing is to abstract this expensive component into a new variable, say  $i'$ , and to update the variable whenever when the subcomponents  $i$  and  $c$  are modified. An optimizing component can then generate the appropriate code for each of these modifications. For the above example, the compiler may be able to generate the program

```

i' := 0;
for (i:=0;i < n;i++) {
    f := f + i';
    i' := i' + c;
}
```

where multiplications are replaced by additions.

More generally, the fundamental idea behind finite differencing is to abstract expensive computations by abstract data types that maintain a state and whose operations (called derivatives) specify how to update this state when one of the subcomponents of the abstraction is modified. This idea applies of course to numerical expressions and Paige [14] showed how it could be generalized to complex data structures.

The implementation of invariants can be viewed as a generalization of the finite differencing approach. The finite differencing approaches that we are aware of correspond to our static invariants. This comes from the fact that these techniques were used in the context of optimizing compilers and thus it was necessary to perform the transformations at compile time. LOCALIZER generalizes this approach to dynamic invariants that are critical for many local search algorithms. As mentioned, dynamic invariants require to interleave the planning (or compilation in this context) and execution phases.

### 6.5. *Programming with Invariants*

Finite differencing was also used by Paige [15] to propose a new syntactic construct called invariant.<sup>3</sup> The motivation behind the paper was the recognition that a number of efficient, but involved, programs are in fact incremental versions of much simpler algorithms. For instance, heap sort can be viewed as a version of selection sort where the minimum element is maintained incrementally. The paper then suggests to use invariants to maintain these data structures incrementally. Here, an invariant is a syntactic construct that defines a state and whose operations are derivatives that specify how to update the state when some other variables are modified. In other words, Paige's invariants are abstractions to construct what we called invariants in LOCALIZER. A pre-processor then rewrites these invariants into traditional code through successive transformation.

It is interesting to observe that LOCALIZER shares the same motivations and address the same issues with related concepts. There are however some important differences. First, at the conceptual level, LOCALIZER supports a rich set of invariants without requiring user intervention. Our basic motivation here is that the definition of these invariants is difficult and should be automated as much as possible. Second, at the implementation level, Paige's invariants are static (they are rewritten into conventional code at compile time) and dynamic invariants are not supported. As mentioned, dynamic invariants are critical for achieving good performance on a variety of local search algorithms.

The idea of letting users define their own invariants is however very appealing, since it may make the language more extensible. It is particularly attractive if the techniques of LOCALIZER are integrated in an object-oriented library. It is an interesting open issue to determine how this can be achieved for dynamic invariants.

### 6.6. *INC: An Incremental Programming Language*

Another related research, once again from the programming language field, is INC [27], a language for incremental computations. INC is a language that allows to write non recursive functions in the style of functional programming. These functions are transformed by the INC compiler into an incremental algorithm which, given a variation of the input, produces a new output. As mentioned, INC uses functional notation and its primitives (e.g., FILTER, TUPLIFY, EQUIJOIN) are closely related to our normalized invariants. This is of course not surprising since bags and tuples are used as the main data structures. The implementation of these operations are based on finite differencing [14].

Once again, there are fundamental differences between INC and LOCALIZER. At the conceptual level, there are three main differences. First, invariants in LOCALIZER are at a much higher level of abstraction than INC functions. Invariants are generally close the data used in informal descriptions of local search algorithms. INC functions, on the other hand, are roughly at the level of our elementary invariants. Since our goal is to provide high-level abstractions for local search, the transformation from invariants to normalized invariants is best left to the compiler, especially since this translation can be performed effectively. Second, and equally important, INC has no counterpart for our

elementary dynamic invariants, e.g., `element`. This is an important limitation since these invariants are fundamental in practical applications. Third, INC has no support to define recursively defined data structures as in the job-shop application. It is of course possible to extend INC to remedy these limitations. High-order functions could be introduced to support dynamic invariants and a fixpoint operator can be added to support recursively defined data structures. However, it is not clear that the INC compiler could recover enough of the structure of the application to obtain the same efficiency as LOCALIZER. Also, implementing these extensions in full generality may be challenging and not needed in practice. At the implementation level, of course, INC supports only static invariants. As far as we know, no experimental results were reported on significant applications.

In summary, invariants seem to provide a much better expressiveness/efficiency trade-off<sup>4</sup> than INC functions to support local search algorithms and extensions of INC are likely to be too general and thus probably inefficient or ad-hoc.

### 6.7. Incremental Algorithms

Finally, it is useful to link our research to the large body of research on incremental algorithms. It is not our goal to review this research in full detail but rather to explain what we mean by efficient incremental algorithms in this paper.

The analysis of algorithms in the paper is in the spirit of the work of Ramalingam and Reps [20] and is expressed in terms of the size of the variations in input and output. Our basic motivation in analyzing algorithms in this way comes from the fact that this analysis is much more informative in this context than a traditional worst-case online analysis. Indeed, the objective of incremental algorithms in LOCALIZER is to compute a new output for a, generally small, variation in the input. However, sometimes a small variation in the input may induce a dramatic variation in the output so that an analysis taking into account only the input will not be informative, even in an amortized sense. Note that, with this in mind, an incremental algorithm that turns out to be linear in the input/output variations is optimal since all the new bits in the input must be seen and at least all the new bits in the output must be written. Almost all our elementary invariants have optimal implementations in this sense in the current implementation of LOCALIZER. Similarly, high-level invariants such as `distribute` and `dcount` are optimal.

It is also interesting to discuss global analysis. A global analysis of the model for the simple job-shop scheduling statement shows that LOCALIZER is running in time  $O(\delta)$  where  $\delta$  is the size of the change in input and output. This is the same bound as the incremental algorithm for single source shortest path proposed in [20]. The same result would not hold when the degree for the vertices is not bounded. This is one of the primary motivations to consider global invariants. Global invariants not only increase the expressiveness of the language but also make it possible to obtain better incremental behaviour. We expect that research on global invariants will be as important as research on global constraints in constraint programming.

## 7. Conclusion

The main contribution of this paper is to show that local search can be supported by modeling languages to shorten the development time of these algorithms substantially, while preserving the efficiency of special-purpose algorithms. To substantiate this claim, we presented a progress report on the modeling language LOCALIZER, introduced in [13]. LOCALIZER statements are organized around the traditional concepts of local search and may exploit the special structure of the problem at hand. The main conceptual tool underlying LOCALIZER is the concept of invariants which make it possible to specify complex data structures declaratively. These data structures are maintained incrementally by LOCALIZER, automating one of the most tedious and error-prone parts of local search algorithms. Experimental results indicate that LOCALIZER can be implemented to run with an efficiency comparable to specific implementations.

Our current research focuses on building higher-level data structures to simplify the design of invariants which are the cornerstone of the language. Extending the strategies to accommodate dynamic  $k$ -opt [17], genetic algorithms, constraint techniques [18] are also contemplated. Longer term research will explore how LOCALIZER can be turned into a programming language library to guarantee extensibility and wide applicability for expert users, while preserving the right level of abstraction.

## Acknowledgments

This paper is dedicated to the memory of Paris C. Kanellakis who kept on gently pressuring us to pursue this topic. Thanks to D. McAllester and B. Selman for many discussions on this research and to Costas Bush for implementing the graph-coloring algorithm in C. This research was supported in part by an NSF NYI Award.

## Notes

1. A companion paper for the Operations Research community is oriented around applications and modeling aspects.
2. Note that the algorithmic component may itself be declarative as in constraint logic programming.
3. We became aware of the work of Paige after completion of the first implementation but it is interesting to observe that the same name was used for two different, but related, concepts.
4. Note that INC has as transitive closure operator that is an extension that is being implemented in LOCALIZER

## References

1. Bisschop, J., and Meeraus, A. (1982). On the Development of a General Algebraic Modeling System in a Strategic Planning Environment. *Mathematical Programming Study*, 20:1–29.
2. Borning, A. (1981). The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transaction on Programming Languages and Systems*, 3(4):353–387.

3. Caseau, Y., and Laburthe, F. (1995). Claire: A Brief Overview. Technical report, LIENS, École normale supérieure.
4. Colmerauer, A. (1990). An Introduction to Prolog III. *Commun. ACM*, 28(4):412–418.
5. Dell’Amico, M., and Trubian, M. (1993). Applying Tabu Search to the Job-Shop Scheduling Problem. *Annals of Operations Research*, 41:231–252.
6. Dincbas, M., Van Hentenryck, P., Simonis, H., Aggoun, A., Graf, T., and Berthier, F. (1988). The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December.
7. Fourer, R., Gay, D., and Kernighan, B. W. (1993). *AMPL: A Modeling Language for Mathematical Programming*. The Scientific Press, San Francisco, CA.
8. Henz, M., Smolka, G., and Würtz, J. (1993). Oz—A Programming Language for Multi-Agent Systems. In Ružena Bajcsy, editor, *13th International Joint Conference on Artificial Intelligence*, volume 1, pages 404–409, Chambéry, France, 30 August–3 September. Morgan Kaufmann Publishers.
9. Jaffar, J., Michaylov, S., Stuckey, P. J., and Yap, R. (1992). The CLP( $\mathfrak{R}$ ) Language and System. *ACM Trans. on Programming Languages and Systems*, 14(3):339–395.
10. Johnson, D., Aragon, C., McGeoch, L., and Schevon, C. (1989). Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research*, 37(6):865–893.
11. Johnson, D., Aragon, C., McGeoch, L., and Schevon, C. (1991). Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning. *Operations Research*, 39(3):378–406.
12. Michel, L. (1998). *Localizer: A Modeling Language for Local Search*. PhD thesis, Brown University, October.
13. Michel, L., and Van Hentenryck, P. (1997). Localizer: A Modeling Language for Local Search. In *Second International Conference on Principles and Practice of Constraint Programming (CP’97)*, Linz, Austria, October.
14. Paige, R. (1981). *Formal Differentiation*. PhD thesis, Dept. of Computer Science, New York University.
15. Paige, R. (1986). Programming with Invariants. *IEEE Software*, January:56–69.
16. Paige, R., and Koenig, S. (1982). Finite Differencing of Computable Expressions. *ACM Transactions on Programming Languages and Systems*, 4(3):402–454.
17. Papadimitriou, C. H., and Steiglitz, K. (1982). *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ.
18. Pesant, G., Gendreau, M., and Rousseau, J. M. (1997). Genius-cp: A Generic Single-Vehicle Routing Algorithm. In Gert Smolka, editor, *Principle and Practice of Constraint Programming—CP97*, Lecture Notes in Computer Science, pages 420–434. Springer, October.
19. Puget, J. F. (1994). A C++ Implementation of clp. In *Proceedings of SPICIS*, November.
20. Ramalingam, G., and Reps, T. (1991). On the Computational Complexity of Incremental Algorithms. Technical report, University of Wisconsin-Madison.
21. Schrage, L. E. (1997). *Optimization Modeling with LINDO*. Duxbury, 5<sup>th</sup> edition, February.
22. Selman, B., Levesque, H., and Mitchell, D. (1992). A New Method for Solving Hard Satisfiability Problems. In *AAAI-92*, pages 440–446.
23. Stuckey, P., and Tam, V. (1996). Models for Using Stochastic Constraint Solvers in Constraint Logic Programming. In *PLILP-96*, Aachen, August.

24. Sutherland, I. E. (1963). SKETCHPAD: A Man-Machine Graphical Communication System. MIT Lincoln Labs, Cambridge, MA
25. Van Hentenryck, P. (1999). *OPL: The Optimization Programming Language*. The MIT Press, Cambridge, Mass.
26. Van Hentenryck, P., Michel, L., and Deville, Y. (1997). *Numerica: a Modeling Language for Global Optimization*. The MIT Press, Cambridge, Mass.
27. Yellin, D. M., and Strom, R. E. (1988). INC: A Language for Incremental Computations. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, pages 115–124, Atlanta, Georgia, 22–24 June. *SIGPLAN Notices* 23(7), July.