Localizer

Laurent Michel and Pascal Van Hentenryck

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-98-02
March 1998

LOCALIZER

Laurent Michel and Pascal Van Hentenryck
Brown University
Box 1910
Providence, RI 02912
{ldm,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 domain-specific 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 schedulingq show the feasibility of the approach.

1 Introduction

Most combinatorial search problems are solved through global or local search. In global 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 neighborhood 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 [2] and NUMERICA [10] to constraint programming languages such CHIP, ILOG SOLVER, CLP($\Re$), PROLOG-IV, and OZ 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., [9]). 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 [4]), 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 [5] is a domain-specific 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. 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.[1] 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. Finally, Section 6 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. 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      if value(s) < bestBound then
5.1       bestBound := value(s);
5.2       best := s;
```

[1] A companion paper for the Operations Research community is oriented around applications and modeling aspects.

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

```
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

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 and Claire 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 first-order boolean formula expressed in disjunctive normal form. The input is given as a set of clauses,

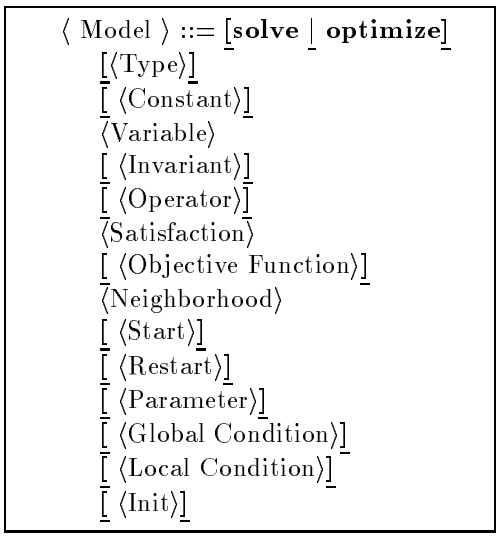

Figure 2: The Structure of LOCALIZER Statements

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 model considered for Boolean satisfiability is based on the GSAT algorithm by Selman et al. in [8], where the local search moves consist of flipping the truth value of an atom.

A local improvement model for Boolean satisfiability is described in Figure 3. In the model, 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 model 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

```
        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:
    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 Model for Boolean Satisfiability

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 model.

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

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

```
nbtl : array[ i in 1..m] of int = sum(i in cl[i].p) a[j] + sum(j in cl[i].n) !a[j];
```

in the Boolean satisfiablity model specifies that *nbtl[c]* is equal to the sum of all true positive atoms and all false negative atoms in clause c, for all clauses in $1..n$. 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, *nbtl[c]* is updated in constant time.

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

```
nbClauseSat : int = sum(i in 1..m) (nbtl[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 evaluated 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};
NEmtpy : {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 model implementing an algorithm in [4]. The graph-coloring problem amounts to finding the smallest number of colors to label a graph such that 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

NEmtpy : {**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 **size***(C[i])* > *0* evolves over time.

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 model 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;

```
        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 Model for Boolean Satisfiability

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 model, depicted in Figure 4, captures the essence of the GSAT algorithm.

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 model is given in Figure 5.

The model illustrates also several new features of LOCALIZER. The **Operator** section describes two procedures which are used 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 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 → t[v] := trial;
```

```
        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();
```

Figure 5: A Simulated Annealing Model for SAT

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 model 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 model 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
```

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 models 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

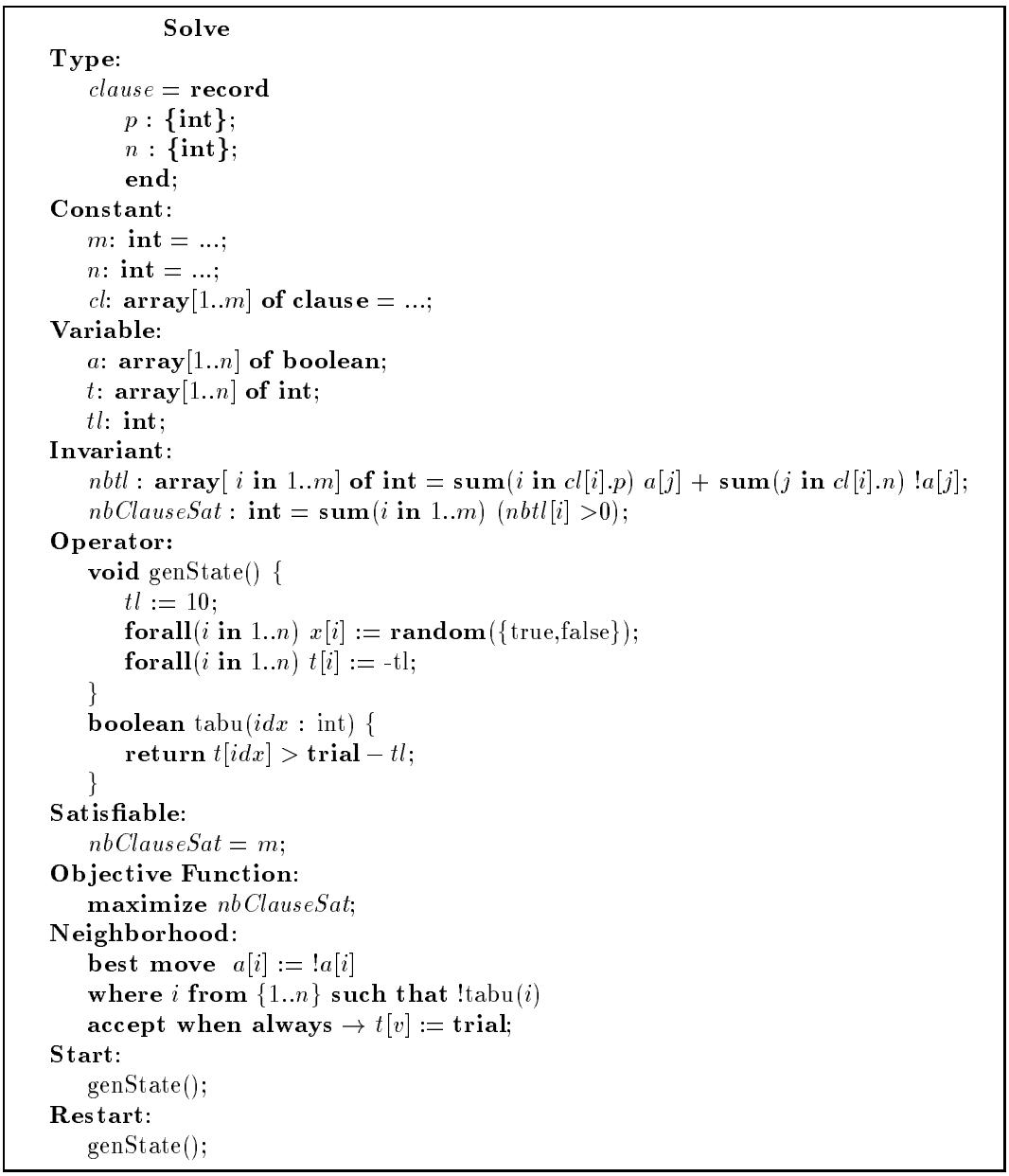

Figure 6: A Tabu Search Model for Boolean Satisfiability

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] >= 0;
```

evaluates the condition $gain[i] >= 0$ 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

```
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];
    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];
    nbClauseSat : int = sum(i 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 model.

The GSAT model 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 = max(i in 1..n) gain[i];
    Candidates : {int} =
        {i : int | select i from 1..n where gain[i] = maxGain and gain[i] >= 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 model is depicted in Figure 7. Of course, the same transformation can be performed for the tabu search model.

```
        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..m ] 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 Model 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 stating the model. 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 initializationmust match the type of the declaration. Offline initializations can be used for arbitrary types. For instance, the Boolean satisfiability models 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}>];

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 [10]. 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 model. 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: **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** $cl[c].n$};

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.

3.3 Variables

Variables are of course fundamental in LOCALIZER, since they define the state of the computation. Variables are declared in the same ways 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 model where $C[i]$ refers to another invariant whose type is $\{int\}$ (set of integers).

⟨expr⟩ ::= ⟨expr⟩ ⟨op⟩ ⟨expr⟩
::= **if** ⟨expr⟩ **then** ⟨expr⟩ **else** ⟨expr⟩
::= - ⟨expr⟩
::= ⟨aggr⟩**(** ⟨identifier⟩ **in** ⟨Set Body⟩ **)** ⟨expr⟩
::= **(** ⟨expr⟩ **)**
::= *constant literal*
::= ⟨Set Expr⟩
::= ⟨expr⟩ **.** ⟨identifier⟩
::= ⟨expr⟩ [⟨expr$_1$⟩ , . . . ,⟨expr$_n$⟩]
::= ⟨function⟩ **(** ⟨expr$_1$⟩ , . . . ,⟨expr$_n$⟩ **)**
::= **{** ⟨Set Body⟩ **}**
::= **{** ⟨expr⟩ [, ⟨expr⟩] **}**
::= **{** ⟨identifier⟩ **:** ⟨typeId⟩ | [⟨select⟩] **}**

⟨op⟩ ::= [**and** | **or** | < | > | = | <= | >= | <> | + | - | * | / | % | ↑ | **union**]

⟨aggr⟩ ::= [**sum** | **prod** | **min** | **max** | **argmin** | **argmax**]

⟨Set Body⟩ ::= ⟨expr⟩ **..** ⟨expr⟩ | ⟨expr⟩
::= ⟨expr⟩ **..** ⟨expr⟩
::= ⟨expr⟩

⟨select⟩ ::= **select** ⟨identifier⟩ **from** ⟨range⟩ [**where** ⟨expr⟩]
::= **select** ⟨identifier⟩ **from** ⟨expr⟩ [**where** ⟨expr⟩]

⟨range⟩ ::= ⟨expr⟩ **..** ⟨expr⟩

Figure 8: Expression syntax

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

Figure 9: Primitive Functions

$Empty$: {int} := { i : int | **select** i **from** 1..n **where size**($C[i]$)=0};
B : **array**[k **in** 1..n] **of** {$Edge$} := { $< i, j >$: $Edge$ | **select** i **from** $C[k]$
select j **from** $C[k]$
where $A[i, j]$ };

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 : **array**[O] **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:

$$\begin{array}{lcl} B = distribute(A, I, O) & \Leftrightarrow & B[i] = \{j \in I | A[j] = i\}\ \forall i \in O \\ B = dcount(A, I, O) & \Leftrightarrow & B[i] = \textbf{size}(\{j \in I | A[j] = i\})\ \forall i \in O \end{array}$$

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 models of Figures 5 and 6.

⟨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⟩ ::= **{** ⟨identifier$_1$⟩ **=** ⟨expr$_1$⟩ **;** ... **;** ⟨identifier$_n$⟩ **=** ⟨expr$_n$⟩ **}**

Figure 10: The Syntax of Statements

3.6 Neighborhood

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

move $op(x_1, \ldots, x_n)$
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 $op(x_1, \ldots, x_n)$, the transformation of the current state into one of its neighbors. The second part starting with the **where** keyword 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

⟨neighorhoood⟩ ::= **move** $op(x_1, \ldots, x_n)$
where
⟨letExpr$_1$⟩
...
⟨letExpr$_n$⟩
[**accept when** ⟨AcceptanceCriterion⟩]
⟨letExpr⟩ ::= ⟨identifier⟩ **from {** ⟨Set Body⟩ **}** [⟨optClause⟩]
::= ⟨identifier⟩ **from {** ⟨expr$_1$⟩ **,** ... **,** ⟨expr$_n$⟩ **}** [⟨optClause⟩]
::= ⟨identifier⟩ **from {** ⟨identifier⟩ **:** ⟨type⟩ | ⟨select$_1$⟩ ... ⟨select$_n$⟩**}** [⟨optClause⟩]
::= ⟨identifier⟩ **from** ⟨lvalue⟩ [⟨optClause⟩]
::= ⟨identifier⟩ = ⟨expr⟩

Figure 11: The Syntax of **move** Instructions

⟨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 keywords **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 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 model 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 model 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 models 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.

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:

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

where c is a constant, $x, y, z, x_1, \ldots, x_n$ 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**, and **min**. Relations return 1 when true and 0 otherwise. An invariant

$$x := element(e, x_1, \ldots, x_n)$$

assigns to x the element in position e in the list $[x_1, \ldots, x_n]$. 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), $def(I)$ denotes x while $exp(I)$ 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 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., [1]).

This section describes static invariants which impose a static restriction. Although this restriction may seem strong, it accommodates many models for applications such as satisfiability and graph coloring to name a few. 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 1 The topological constraints of invariant I are defined as follows:

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

Definition 2 The topological constraints of a set of invariants $\mathcal{I}$ denoted by $tc(\mathcal{I})$ is $\bigcup_{I \in \mathcal{I}} tc(I)$.

Definition 3 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.

```
procedure execute(I,M,t)
begin
    Q := {⟨x, I⟩|x ∈ M ∧ I ∈ invariants(x, I)}
    while Q ≠ ∅ do
        ⟨x, I⟩ := POP(Q,t);
        propagate(x,I,I,Q′);
        Q := Q ∪ Q′;
    endwhile
end

function POP(Q,t)
    pre: Q is not empty
    post: ⟨x, I⟩ ∈ Q such that ∀⟨x′, I′⟩ ∈ Q : t(I′) ≥ t(I)
```

Figure 12: The Execution Phase for Static Invariants

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

$$x := sum(x_1, \ldots, x_n)$$

is performed by a procedure

```
procedure propagate(x_i,x := sum(x_1, ..., x_n),I,Q)
begin
    x^c := x^c + (x_i^c − x_i^o);
    if x^c ≠ x^o then
        Q := invariants(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 $\langle x_j, x := sum(x_1, \ldots, x_n)\rangle$ may need to be propagated.

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

$$x := element(e, y_1, \ldots, y_n)$$

The static topological constraint for this invariant is

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

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:

$$\begin{array}{lcl} start_3 & := & max(p, d) \\ p & := & element(prec_3, end_1, \ldots, end_n) \\ d & := & element(disj_3, end_1, \ldots, end_n) \end{array}$$

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

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

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

$$x := element(e, y_1, \ldots, y_n)$$

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

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

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 $l : \mathcal{V} \to \mathcal{N}$ and $t : \mathcal{V} \to \mathcal{N}$ and of two sets of constraints.

Definition 4 The level constraints associated with an invariant I and denoted by $lc(I)$ are defined as follows:

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

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 5 The level constraints associated with a set of invariants $\mathcal{I}$ and denoted by $l(\mathcal{I})$ is simply $\bigcup_{I \in \mathcal{I}} lc(I)$.

Definition 6 A set of invariants $\mathcal{I}$ is serializable if there exists an assignment $l : \mathcal{V} \to \mathcal{N}$ satisfying $lc(\mathcal{I})$.

A serializable set of invariants can be partitioned into a sequence $< \mathcal{I}_0, \ldots, \mathcal{I}_p >$ 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 7 Let S be a computation state and let $S(x)$ denote the value of x in S. The static constraints associated with an invariant I wrt to S, denoted $sd(I, S)$ is defined as follows:

$$\begin{array}{lcl} sd(x := c, S) & = & \{t(x) = 0\} \\ sd(x := y \oplus z, S) & = & \{t(x) = max(t(y), t(z)) + 1\} \\ sd(x := \prod(x_1, \ldots, x_n, S) & = & \{t(x) = max(t(x_1), \ldots, t(x_n)) + 1\} \\ sd(x := element(e, x_1, \ldots, x_n), S) & = & \{t(x) = t(x_{S(e)}) + 1\} \end{array}$$

Definition 8 The static constraints associated with a set of invariants $\mathcal{I}$ wrt to a state S, denoted by $sd(\mathcal{I}, S)$ is simply $\bigcup_{I \in \mathcal{I}} sd(I, S)$.

Definition 9 A set of invariants $\mathcal{I}$ is static wrt to a state S if there exists an assignment $t : \mathcal{V} \rightarrow \mathcal{N}$ satisfying $sd(\mathcal{I}, S)$.

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.

Definition 10 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 $< \mathcal{I}_0, \ldots, \mathcal{I}_p >$;
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.

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.

```
procedure execute(I,M)
begin
    < I_0, ..., I_p > := serialize(I);
    for( i := 0; i <= p; i++) do
        t := plan(I_i);
        execute(I_i,M,t);
    endfor
end
```

Figure 13: The Execution Algorithm for Dynamic Invariants

5 Experimental results

This section summarizes some preliminary results on the implementation of LOCALIZER (about 35,000 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 [8]. Table 1 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 [8], 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 model 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.

5.2 Graph Coloring

Graph coloring was the object of an extensive experimental evaluation in [4] 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 [4], 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

	V	C	I	L	G	R
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
Av.						3.38

Table 1: GSAT: Experimental Results

consistent with those in [4]. 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 model is about one page long.

Quality of the Solutions Table 2 describes the quality of the coloring found by LOCALIZER. These results agree with those of the C implementation and with those reported in [4]. Each set of rows corresponds to a class of graphs and to 100 executions of LOCALIZER on graphs from this class. The rows in each set report on the various values found by LOCALIZER on these graphs and their frequencies. The columns report the number of vertices, the density of the graph, the size factor sf used in the experiments, the number of colors found by some solution, and the frequency of colorings of this quality. For instance, the first set of rows 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 3 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. 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.

	Vertices	*Density*	*Size Factor (sf)*	*Colors*	*Frequency*
random	125	50	3	19	92
				20	8
random	250	50	4	31	1
				≤ 32	8
				33	43
				34	44
				≥ 35	4
random	500	50	4	55	4
				56	54
				57	41
				≥ 58	1
random	125	10	1	6	48
				7	52
random	250	10	1	9	27
				10	70
				11	3
random	500	10	2	15	1
				16	79
				17	20
random	125	90	1	≤ 44	7
				45	30
				46	30
				≥ 47	33
random	250	90	1	≤ 78	17
				79	25
				≥ 80	58
random	500	90	1	≤ 143	30
				144	22
				≥ 145	48
cooked	125		4	9	100
cooked	250		1	15	100
cooked	500		2	25	71
				≥ 25	29

Table 2: Graph Coloring: Quality of the Solutions

	Vertices	*Density*	*Size Factor (sf)*	LOCALIZER (L)	*C Implementation* (C)	*L/C*
random	125	50	3	78.3	18.9	4.50
random	250	50	4	82.8	18.4	4.50
random	500	50	4	633.7	123.4	5.10
random	125	10	1	22.5	4.8	4.60
random	250	10	1	109.2	22.02	4.96
random	500	10	2	159.8	28.8	5.50
random	125	90	1	16.18	4.53	3.56
random	250	90	1	49.32	8.89	5.54
random	500	90	1	162.7	29.6	4.88
cooked	125		4	22.09	4.18	5.28
cooked	250		1	37.22	7.79	4.77
cooked	500		2	240.3	49.9	4.80
average						4.82

Table 3: Graph Coloring: Efficiency of LOCALIZER

5.3 Graph Partitioning

The problem has been studied experimentally in [3] and, once again, the experiments reported here are based on a similar setting. Table 4 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, SF is the size factor and the remaining two were described previously.

Table 5 compares LOCALIZER with the results reported in [3].

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 model used for these experiment 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 neighorhood 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 [6]. The parameters were defined as follow:

- 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 [6].
- Contrary to [6], the model does not use a restarting strategy.

Graph			Results (T=10,TF=0.95,SF=16,chPerc=2%,Cutoff=10%)						
Class	*Vertices*	*Density*	*SB*			*Freq.*			LOCALIZER
random	124	2	11–13	14–16	17–19	33	38	29	2.22
random		4	55–59	60–65	66–77	30	34	36	2.40
random		8	159–174	175–190	191–	23	53	24	3.24
random		16	481–560	561–640	641–	36	45	19	4.05
random	250	1	20–24	25–29	30–35	19	22	59	4.68
random		2	92–106	107–121	122–131	12	42	36	5.10
random		4	324–343	344–363	364–380	38	43	19	6.50
random		8	828–877	878–927	928–	37	40	23	9.98
random	500	0.5	48–54	55–59	60–66	15	43	42	10.08
random		1	219–231	232–244	245–256	17	52	31	10.76
random		2	637–661	662–686	686–718	10	15	75	14.44
random		4	1661-1701	1702–1741	1742–1824	22	58	20	20.67
random	1000	0.25	90–103	104–118	119-126	41	52	7	19.99
random		0.5	439–455	456–475	476–503	36	43	21	22.62
random		1	1326–1357	1358–1397	1398–1427	33	51	16	29.97
random		2	3253–3319	3320–3394	3394–3466	11	37	52	40.10
Class	*Vertices*	$n\pi d^2$	*SB*			*Freq.*			LOCALIZER
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

Table 4: Graph Partitioning: Experimental Results

Graph			Results (T=10,TF=0.95,SF=16,chPerc=2%,Cutoff=10%)				
Class	*Vertices*	*Density*	*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

Table 5: Graph Partitioning: Comparison Results

Table 6 reports the preliminary results. These results cannot be really compared with the results of [6], since the neighborhood used in their experiment is $(RN1 \cup RN2)$ while the model used here relies on $N1$ alone. The time reported correspond to the algorithm termination (once the 12000 iterations are spent), not the time required to produce the best or the optimal solution for the first time. Note also that the 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 produce the optimum solution in about 0.5 seconds. Interestingly enough, even the simple neighborhood $N1$ does quite well and finds the optimum solution for 18 benchmarks (out of 28 benchmarks). The table also reports a coarse histogram that summarizes the frequencies of apparition for the solutions. These frequencies were obtained based on a serie of 100 experiment for each benchmark.

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

6 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 domain-specific language LOCALIZER, introduced in [5]. 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 underly-

Benchmark				Results (MD=12000,MS=1,Neighborhood=N1)									
Name	*Job*	*M.*	*Opt*	*Ranges*				*Freq.*				Loc	*Avg.sol.*
LA01	10	5	666	666	-	-	-	100	-	-	-	25.8	666
LA02	10	5	655	655	656–661	662–668	669–676	11	27	51	11	26.5	663.3
LA03	10	5	597	604	605–614	615–625	625–635	2	19	62	17	26.5	619.7
LA04	10	5	590	590	591–595	596–601	602–606	10	28	58	4	25.5	596.2
LA05	10	5	593	593	-	-	-	100	-	-	-	25.3	593
LA06	15	5	926	926	-	-	-	100	-	-	-	28.6	926
LA07	15	5	890	890	-	-	-	100	-	-	-	31.1	890
LA08	15	5	863	863	-	-	-	100	-	-	-	30.6	863
LA09	15	5	951	951	-	-	-	100	-	-	-	29.5	951
LA10	15	5	958	958	-	-	-	100	-	-	-	28.3	958
LA11	20	5	1222	1222	-	-	-	100	-	-	-	35.2	1222
LA12	20	5	1039	1039	-	-	-	100	-	-	-	33.9	1039
LA13	20	5	1150	1150	1151–1152	1153–1155	1156–1159	97	0	0	3	34.2	1150.2
LA14	20	5	1292	1292	-	-	-	100	-	-	-	32.4	1292
LA15	20	5	1207	1207	-	-	-	100	-	-	-	38.0	1207
MT6	6	6	55	55	-	-	-	100	-	-	-	15.6	55
MT10	10	10	930	936	937–956	957–977	978–997	1	24	60	15	39.9	966
MT20	20	5	1165	1165	1166–1202	1203–1240	1241–1278	2	80	17	1	38.9	1189.3
ORB1	10	10	1059	1073	1074–1100	1101–1128	1129–1157	1	8	45	46	40.5	1124.3
ORB2	10	10	888	889	890–898	899–908	909–918	4	34	50	12	38.5	900.8
ORB3	10	10	1005	1021	1022–1093	1094–1166	1167–1238	1	95	1	3	42	1061.6
ORB4	10	10	1005	1011	1012–1027	1028–1044	1045–1060	3	18	60	19	37.1	1034
ORB5	10	10	887	895	896–911	912–928	929–946	3	34	44	19	40.8	917.6
ORB6	10	10	1010	1013	1014–1029	1030–1046	1047–1063	1	17	61	21	40.5	1038.9
ORB7	10	10	397	397	398–404	405–412	413–419	1	17	69	13	41.6	408.8
ORB8	10	10	899	919	920–940	941–962	963–983	1	35	53	11	42.7	948.1
ORB9	10	10	934	944	945–955	956–967	968–978	2	32	46	20	36.4	960.8
ORB10	10	10	944	944	945–963	964–983	984–1002	1	46	50	3	40	965.2

Table 6: Job-Shop Scheduling: Experimental Results

ing 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 [7], genetic algorithms, constraint techniques 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.

References

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

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

[3] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning. *Operations Research*, 37(6):865–893, 1989.

[4] D. Johnson, C. Aragon, L. McGeoch, and C. Schevon. Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning. *Operations Research*, 39(3):378–406, 1991.

[5] L. Michel, P. Van Hentenryck. Localizer: A Modeling Language for Local Search. In *Second International Conference on Principles and Practice of Constraint Programming (CP'97)*, Linz, Austria, October 1997.

[6] Marco Trubian Mauro Dell'Amico. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993.

[7] C.H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Prentice-Hall, Englewood Cliffs, NJ, 1982.

[8] B. Selman, H. Levesque, and D. Mitchell. A New Method for Solving Hard Satisfiability Problems. In *AAAI-92*, pages 440–446, 1992.

[9] P. Stuckey and V. Tam. Models for Using Stochastic Constraint Solvers in Constraint Logic Programming. In *PLILP-96*, Aachen, August 1996.

[10] P. Van Hentenryck, L. Michel, and Y. Deville. *Numerica: a Modeling Language for Global Optimization.* The MIT Press, Cambridge, Mass., 1997.