SaLSA Specification language for search algorithms

Yves CASEAU
François LABURTHE

LIENS - 97 - 11

SaLSA Specification language for search algorithms

Yves CASEAU
François LABURTHE

LIENS - 97 - 11

November 1997

Laboratoire d'Informatique de l'Ecole Normale Supérieure
45 rue d'Ulm 75230 PARIS Cedex 05

Tel : (33)(1) 44 32 30 00
Adresse électronique : laburthe@dmi.ens.fr , caseau@dmi.ens.fr

SaLSA: a Specifiction Language for Search Algorithms

Yves Caseau, François Laburthe

1. Abstract

This paper presents a language and a system for expressing complex search algorithms. Algorithmic components involving search (non-determinism and backtracking) often appear as sub-routines of complex optimization algorithms. Such algorithms usually combine local and global search, each method being specialized (using dedicated heuristics, branching procedures or neighborhoods) and complex (combining various backtracking or search strategies). Hence, the standard programming process of such algorithms is usually tedious and error-prone. By specifying search procedures with small terms of the SaLSA language, we propose to raise the level of abstraction of hybrid algorithm descriptions. Such a language, together with its associated compiler and environment, offers new opportunities for improving the development time, maintenance cost and reliability of combinatorial optimization software.

2. Motivations

Large-scale industrial combinatorial problems are seldom solved by using in a straight-forward manner a well-known algorithm from the Operations Research literature. Most of the times, because of the specificity of the problem or because of its intrinsic complexity (problems involving several types of constraints, or several criteria to optimize), the programmer has to use, in order to tackle such problems, several algorithmic components, combine them, and design a hybrid algorithm. Hybrid algorithms thus combine different optimization paradigms such as local optimization, linear programming, semi-definite programming, constraint propagation, branch and bound, etc...

Unfortunately, the design of hybrid algorithms is not (yet) a well understood engineering process and few tools are available to the programmer : each algorithmic technology usually comes with its own language (a modeling language for local optimization, CLP for constraint programming, a linear package for LP, ...), but there is usually little available to combine such components. In some case, (coarse grain cooperation between the algorithms), assembling the components is easy (for example, if the components operate one after the other, the overall algorithm is only a sequence of calls to distinct procedures). In other cases, when the algorithms are intimately interleaved, the cooperation may be much harder to organize.

Many search algorithms belong to this category of fine-grain algorithmic cooperation (in particular for heuristics enriched with backtracking -e.g. LDS- or algorithms featuring constraint propagation and local optimization). However, the overall scheme (as described in textbooks) of search algorithms is simple :

- local search is based on a visit of a series of neighborhoods
- global search is based on the exploration of choice points, with a particular backtracking strategy

Hence, local and global search may be described by means of local structures (neighborhoods or choice points) and visit mechanisms (iterate moves / backtracking strategy). This suggests that much of the specificity of search algorithms is in the control of the algorithm. This presentation will indeed focus on specifying the flow of control of hybrid search algorithm, and will set aside other algorithmic issues (which may be utterly important), such as incremental consistency maintenance, constraint propagation, etc. Anyhow, we feel that giving a concise description of the control mechanisms in a hybrid algorithm is an improvement over state of the art of hybrid algorithm descriptions and that the global understanding of the behavior of the system that it offers is highly valuable. Considering the search mechanism as a control structure, there is no reason to express search algorithms only within tools/languages dedicated to one particular programming paradigm (such as constraint propagation). Instead, such a standpoint suggests the enrichment of a full-fledged programming language with new control structures dedicated to search (see for example [AC 97].

Within today's languages available for combinatorial optimization, the engineering process of such specialized tree search algorithms is usually a tedious task. In fact the « best common platform » supporting all paradigms is usually :

- either a Prolog-like logic language with an implicit search strategy. In this case, the programmer may try to specialize the search algorithm by encoding some control in the logic formulation. Because this process amounts to modifying a clean specification of a problem with some logic intended only for resolution, it is usually difficult and inelegant.
- or a lower level language with primitives for setting choice points and backtracking (such as a concurrent calculus like Oz or a simple imperative language like Claire)

Both solutions have their drawbacks : the first approach is not open enough towards other optimization paradigms, while the second approach lacks readability and abstraction. Therefore, we propose a new language (SaLSA) dedicated to programming tailored search trees. This language is currently implemented as a Claire preprocessor [CLAIRE], but it could be used in any other system featuring primitives for setting and removing backtrack points (or offering the possibility of creating concurrent processes).

A language such as SaLSA should :

- provide to complex hybrid algorithms simple formal specifications of the flow of control.
- support more creativity from the programmer (simple variations of an algorithm can be easily expressed).

A SaLSA compiler (or pre-processor) should :

- improve the quality of the run-time code (automatically generated)
- support some automatic tuning of a parametric search procedure with respect to a given data-set or some operational constraints (e.g. limited computation time).

Last, a dedicated environment for code generation in SaLSA could offer some highly valuable tools, for producing traces of the search [Mei 95], for monitoring the search (the so-called performance profile of [Zi 93]), for driving the search in an interactive manner [Sch 97].

Last, we should mention that others authors have investigated the combination of local and global search. Pesant et al. [PG97] have proposed a framework for keeping a declarative programming style with hybrid algorithms and Michel and van Hentenryck [MvH97] have proposed a modeling language for local search algorithms (LOCALIZER). This proposal is very elegant, and influenced heavily our presentation of choice points. Our proposal should be fully compatible with the invariant mechanism proposed in [MvH97].

3. Description of the language

3.1 Assembling choice points into terms describing hybrid tree search

SaLSA is intended for the specification of the flow of control in search algorithms. It can be seen as a flow-chart language, dedicated to algorithms with non-determinism. The basic building block of this language is the *choice point*, an operation which considers a set of (mutually exclusive) decisions, and applies only one of them at a time. Hence, as a control mechanism, a choice point leads from one state (a time instant during the computation) to a set of states (since there are several branches). Hence a choice point yields a neighborhood in the space of all states. In a traditional implementation, these branches would be visited sequentially, one after the other. However, in order to account for distributed implementations of search algorithms, we leave unspecified the order of visit between these branches. In a sequential implementation, whenever a failure is detected in a branch, the algorithm « backtracks » (goes back to the father node, which, in turn, may either visit its next son, or if all of them have been visited, may itself backtrack to its own father node). In a distributed implementation, when a failure is detected, the node disappears (and informs its father of its death).

A choice point may actually have two different behaviors, either generating a non empty set of decisions (and hence a strict neighborhood of the current state), or generating an empty set of decisions (yielding an empty neighborhood). In the latter case, applying the choice point has no effect. In this case, we say that the system has come to a *fixpoint* for this choice point.

In order to describe real search algorithms, choice points need to be assembled. Since choice points have two possible behaviors, we consider two composition mechanisms : *leaf-composition* ($\times$) and *fixpoint-composition* ($\Diamond$). For example, the term $C_1 \times C_2$ will first apply the branching mechanism of C_1 before applying that of C_2 in all the branches. As another example, one has $C \Diamond C = C$. Indeed if the current state is at a fixpoint for C, nothing will be done (the environment will be unmodified) and when C is applied a second time, the state is still at a fixpoint for C.

Simple search algorithms generally feature few choice points (sometimes only one). Since, the composing a choice point with itself by fixpoint composition is useless ($C \Diamond C = C$), standard algorithms usually apply $C \times C \times C \times C$ ... C As a shorthand, we offer the power notation C^n ($C^4 = C \times C \times C \times C$). This allows us to describe search algorithms of bounded depth *n*, from which one can either exit at a leaf (after having applied *n* times the choice point C), or earlier, when at some depth *k*, ($k \leq n$), after having already branched *k-1* times, the system has come to a fixpoint for the choice point C.

In some cases, one would be inclined to execute algorithms C^n with very large values of *n*, in order to retain only the fixpoint exits. The construct C* describes this infinite composition of a choice point with himself, until stability (fixpoint state) is reached. The execution of such a term goes through a search tree, in which some branches may be infinite (possibly yielding non terminating program), and in which the exits occur at various depths.

Up to now, we have mentioned « exits » from the search procedure. These are states of the system where the execution of the search algorithm is over. We then need to specify where the flow of control goes. We consider two possibilities : either executing the whole algorithm described by the term until all nodes have come to an exit or stopping the whole algorithm as soon as one state comes to an exit. Those two behaviors are described with the introduction of constructs in the language called *terminators*. There are two of them : *cont* and *exit.* Both terminators can be composed with a term either with $\Diamond$ or $\times$. In each case, they forbid any further composition of this type. For example $T \Diamond exit \Diamond U = T \Diamond exit$ for any pair of terms *T* and *U*. For example, if C is a standard assignment choice point (assigning successively to a variable all values from its domain), $C^* \Diamond$ *exit* will search for solutions and stop at the first one encountered, whereas $C^* \Diamond$ *cont* will search for all solutions, but will return to the initial state of the system (the root node of the search tree) after the search has been performed.

This last example, $C^* \Diamond$ *cont*, is, however, of little interest. Indeed, there is no way of knowing, the number of solutions which have been encountered. One would like, to be able to do some printing or to update some global variables, in each fixpoint-exit of *C**, before continuing the search. In order to offer such a possibility, the composition constructs are extended $\Diamond$ and $\times$ are extended in order to accept as second argument functions as well as terms. The idea is to compose them with functions which have side effects. For example, if *talk* is a function which tells the user about the current solution, we can replace the previous term by a more interesting version $C^* \Diamond$ *talk* $\Diamond$ *cont*, which will evaluate *talk()* in all states which account for fixpoint exits of *C**.

3.2 Creating choice points

Let us now describe how choice points are created. Before stating the grammar for expressing choice points, we start by a simple example (labeling finite domain variables). The syntax should be self explanatory : choice points are named objects (here, the name is *labeling*) and they are described by a set of moves which are simple expressions containing free variables. These free variables are defined after the *with* keyword: some may be bound to a unique value (e.g. *x=some(....)*), others may range over a set of values (e.g. $v \in$ *values(x)*). To highlight choice points in source code, their definition is surrounded by a box.

```
labeling :: Choice
  moves  post(FD, x == v)
    with x = some(x in Variable | unknown?(value,x)),
         v ∈ values(x)
  on failure post(FD, x <> v)
```

The choice point is defined as the control structure which applies moves, one at a time, for all possible assignments of the free variables. The precise operational semantics of choice points is defined in a separate section, but we can consider as a first approximation that the choice point defines a neighborhood of the current state. **Choice***(* **moves** *f(x)* **with** *x* ∈ *g())* applied to the state *s* yields the set of states *{s_1, ..., s_n}* if the evaluation of *g()* returns *{x_1, ..., x_n}* and if state s_i is obtained from state *s* by the evaluation of *f(x_i)*. In the following grammar, we type in bold the reserved keywords and in italic the optional components of a choice point.

Choice points are named objects and are always created with the *Choice* keyword. They may be defined in a parametric manner (here, the parameters are x_1,...,x_i).

```
<ChoicePointName> (x1:<type>,...,xi:<type>) :: Choice
    <ChoicePoint(x1,...,xi)>
```

A choice point with free variables (x_1,...,x_i) is either defined directly with the *moves* keyword, an expression for defining the moves and a set of variable definitions, or it may use a case expression (conditional switch based on typing). They may be defined in a parametric manner (here, the parameters are x_1,...,x_i).

```
<ChoicePoint(x1,...,xi)> ::=
    < moves
          <MovesDef(x1,...,xn)>
          such that <condition(x1,...,xn)>>
          sorted by <order(x1,...,xn)>>
          <vdef({x1,...,xi},xi+1,... xn)>
      on failure <Dexp(x1,...,xn)> > |
    <case <exp(x1,...,xj)> (<type> <ChoicePoint(x1,...,xj)>,
                            ....
                            <type> <ChoicePoint(x1,...,xj)> )
     with <vdef({x1,...,xi},xi+1,... xj)>  >
```

The set of moves considered for a choice point can be defined in three ways : either by explicit enumeration of the expressions to evaluate, separated by the *or* keyword, or by a unique expression containing at least one free variable ranging over a set of values, or by an expression composing other choice points, which we call a SaLSA term.

```
<MovesDef(x1,...,xn)> ::= <Dexp(x1,...,xn)> or <Dexp(x1,...,xn)> .... or <Dexp(x1,...,xn)> |
                          <Dexp(x1,...,xn)> |
                          from <SALSATerm>
```

A choice point hence corresponds to a set of moves, which can be processed in parallel. However, this set may then be modified by two optional lines in the choice point definition. A first line for selecting only a subset of the moves which have been enumerated. This can be done by stating any Boolean condition (which is tested before the move), or by using within a condition the expression *delta(<exp>),* which represent the variation of the integer expression <exp> through the move (this feature is inspired from Localizer) The second optional modification specifies the order in which these moves should be considered.

```
<condition(x1,...,xn)> ::=  <exp(x1,...,xn)> |
                            delta(<exp>) <operator> <exp>
<order(x1,...,xn)> ::= <increasing|decreasing> <exp(x1,...,xn)>
```

The structure of the section for defining the free variables x_{i+1},... x_n from the parameters x_1,...,x_i is described below. Note here that the expressions for defining the values of variable may use set expressions such as {x in E | P(x)}, {f(x) | x in E}, some(x in E | P(x)), some(x in E minimizing e(x)), etc.

```
<vdef({x1,...,xi},xi+1,... xn)> ::= with xi+1 <=,∈> <exp>,
                                         xi+2 <=,∈> <exp(x1)>, ....
                                         xn <=,∈> <exp(x1,...,xn-1)>
```

Finally, we have denoted the expressions describing the moves with the syntax *<Dexp>*. These expressions are indeed special expressions because they may either be distributed across different processes (for parallel computations) or be undone from one branch to the other (for sequential exploration of the branches). For the case of algorithms involving constraint propagation of other mechanisms enforcing some consistency over the set of objects, these expressions are events which trigger some automatic computation (e.g. propagation). In this case, the expressions are richer than simple imperative instructions. To keep the description as declarative as possible, the expressions for moves (which may be standard programming instructions) may consist in posting an event to a solver. The solver is here a black-box with consistency mechanisms, which maintains additional data structures. The way the solver reacts to events is defined somewhere else in the program with a declarative style (by means of constraints, invariants, production rules, concurrent agents, etc...).

```
<Dexp(x1,...,xn)>  ::= post(<solver>, <exp(x1,...,xn)>)
                       <exp(x1,...,xn)>
```

Choice points can be combined together to form SaLSA terms with the two operators (◊ and ×), which have been described earlier and with a few other constructs that will be defined in the next section.

3.3 Composing choice points to produce search trees

This section describes a few other constructs available for expressing search algorithms(except from the two compositions mechanisms).

The first primitive denotes visits to a tree which retain only the best exits (such a procedure is called look-ahead search). Let *T* be a term from which there is only one possibility of exit, either leaf or fixpoint, but not both : for example *C*, C^8 ◊ cont, $C^n \times D^*$* are such valid terms, whereas *C^n, C^n ◊ D** are not. Let *f* be a function of arity 0, *n* be an integer and *r* be a real number between 0.0 and 1.0. Then, the execution of the term *smallest(T,f,n,r)* will perform a look-ahead search of the search tree *T* , will evaluate *f()* in all its exits, will compute the minimal value $f_{\min}$ of these evaluations, and will come back to those exits for which $f() \times r \leq f_{\min}$, keeping at most *n* of them (the smallest ones). For example, if *C* is a choice point, *smallest(C,f,1,1.0)* will go into all branches of *C* , evaluate *f(),* and return to one of the branches yielding the smallest value. A symmetrical primitive, *largest*, is also available.

Two other constructions are also available for expressing global topological conditions on the shape of the search tree. The first one (the operator '/') limits the number of exits of a term by an integer. As for smallest and largest, this applies only to terms having only one possibility for exiting : either *leaf* or *fixpoint*. For example *C* / 4* will behave like *C** until the first four exits (fixpoint exits) and will afterwards stop any further exploration (backtracking to the root node).

Another operator ('!') limits the number of allowed backtracks in the tree. The execution of *T ! n* will be like that of term *T* until exactly *n* backtracks have been performed, in which case, the algorithm will stop any further exploration and backtrack to the root node for *T.*

The two last operator dynamically change the shape of the tree which is being explored : In the term *T where f,* the *where* operator extracts from a given tree definition *T*, only those nodes *n* for which a given condition was verified all along the path from the root node to *n.* Note that this is a generalization to the case of arbitrarily complex SaLSA expressions of the *such that* condition in the definition of one choice point. The next operator, *until,* allows the user to express certain conditions which should cause the search should stop, although the neighborhood has not yet been fully explored. In the term *T until f*, the *until* operator checks the condition *f()* at every inside node of *T* and performs the visit of *T* until an evaluation of *f()*yields true, in which case, it by-passes the rest of the search (deeper in the tree) and considers itself at a fixpoint exit of the term *T.* When the exploration resumes, the exploration resumes until the term comes to a « natural » fixpoint or leaf exit or until it comes to a new fixpoint exit (where *f()* yields true), etc.

3.4 Grammar

In order to complete the grammar that has been show in the previous section, we give the syntax for the SaLSA terms, which may be used either within choice point definitions, or directly within Claire expressions, through the *min, max* or *SOLVE* primitives.

```
<SALSATerm>  ::= <SimpleTerm> |
                 <SimpleTerm> × <SALSATerm> |
                 <SimpleTerm> ◊ <SALSATerm> |
                 <SimpleTerm> × <SALSATerm> ◊ <SALSATerm> |
                 <SALSATerm> / <integer> |
                 <SALSATerm> ! <integer> |
                 <SALSATerm> until <function> |
                 <SALSATerm> where <function> |
                 <terminator>
<SimpleTerm> ::= <ChoicePointName> |
                 <ChoicePointName>^<integer> |
                 <ChoicePointName>* |
                 <smallest | largest>(<SALSATerm>,<function>,<integer>,<percentage>) |
                 <SimpleTerm> × <function> |
                 <SimpleTerm> ◊ <function>
<terminator> ::= cont | exit
<ClaireExpression> ::= <min | max>(<SALSATerm>, <function>) |
                       SOLVE(<SALSATerm>)
```

4. Operational semantics

The purpose of this section is to define formally the behavior of SaLSA expressions. In order to do so, we first need to account for the evaluation of standard expressions of the programming language. If the evaluation of the closed expression (without free variables) *exp* yields the value *val,* the evaluation will be denoted as :

$$(\xi, E) : exp \ \rightarrow \ (\xi', E') : val$$

where the pair *(ξ,E)* denotes the state of the system (the set of existing objects, values of global variables, etc...). In order to express the behavior of SaLSA terms, we will describe a process calculus of distributed computations. This is the reason which the environment of evaluation is separated in two parts : we distinguish a global part of the system, ξ , (which will be shared by all processes) and a local part, E , (each process will have its own version of this environment). Note here, that both parts of the environment may be changed by the evaluation in case of side effects of the considered expression.

To keep the presentation simple, we will consider that all choice points are of the form

```
C :: Choice
  moves  g(x)
    with x ∈ f()
  such that k()
  on failure h(x)
```

which we will denote by *Choice(f,g,h,k).* In fact, if *f()* returns an ordered list of tuples, any choice point definition can be rewritten in such a form.

We start by imposing typing conditions on the four functions *f,g,h,k* used in the choice point definition :

1. *f* is of type *void → set[X],* without side effects
2. *g* is of type *X → {void, ⊥},* with allowed side effects on *E*
3. *h* is of type *X → void,* with allowed side effects on ξ and *E*
4. *k* is of type *void → bool*, without side effects.

where ⊥ is a special value of the language, used for denoting failure (this may correspond to an uncaught exception). After a small amount of rewriting (we introduce an empty term ε such that C × ε =C and C ◊ ε =C, and we rewrite C^n as C×C..... ×C and C^* as the infinite composition C×C×......), we consider that all SaLSA terms can be written in the form of (possibly infinite) binary terms

C × <SALSATerm> ◊ <SALSATerm>,

where C is :
either a choice point,
or a terminator,
or the empty term ε,
or an expression smallest(<SALSATerm>,f,n,r)
or an expression <SALSATerm> / <integer>

We introduce concurrent processes (roughly speaking, there will be one process per node in the search tree), and we will define transitions of a system composed of a global environment and a set of states. Processes will be small objects denoted as $p : \wp$ and will have the following components :

$$p : \wp \left(\begin{array}{l} state : \{start, \exp and, wait, done, failed\} \\ father : \wp \\ sons : set[(< object >, \wp)] \\ \ \ env : set[< object >] \\ code :< SALSATerm > \end{array} \right)$$

Moreover, we will consider a particular process π. The computation of the algorithm represented by the SaLSA term will represented by a sequence of transitions in the process calculus, according to the rules which follow. The transitions will be written:

$$(\xi, \{p_1, ..., p_n\}) \xrightarrow{//} (\xi', \{p_1', ..., p_l'\})$$

where a process p'_j are either distinct from the set of p_i, or it is obtained by changing entries of some process p_i , which will be denoted by

$$p'_j = p_i[field/new \ \ value]$$

We are now ready to state the rules specifying the legal transitions in this system.

$$\frac{\begin{array}{l} p' : \wp(env = E') \\ p : \wp\left(\begin{array}{c} state = start \\ env = E \\ code = T \end{array}\right), \ (\xi, \ p) \xrightarrow{//} (\xi, \ \{p'\}), \ \left(\begin{array}{l} (p'.state = done \ \wedge \ v = \texttt{true}) \\ \vee \ (p'.state = failed \ \wedge \ v = \texttt{false}) \end{array}\right) \end{array}}{(\xi, \ E) : \ \texttt{solve}(T) \ \rightarrow \ (\xi', \ E') : \ v}$$

Rule 1: link with the programming language

$$\frac{(\xi, \ p) \xrightarrow{//} (\xi, \ \{p_1', ..., p_n'\})}{(\xi, \ \{p, p_1, ..., p_k\}) \xrightarrow{//} (\xi, \ \{p_1, ..., p_k, p_1', ..., p_n'\})}$$

Rule 2: expanding one node of the frontier

$$\frac{\begin{array}{l} C = Choice(f,g,h,k) \\ p : \wp\begin{pmatrix} state = start \\ env = E \\ code = C \times T_1 \Diamond T_2 \end{pmatrix},\ (\xi,\ E):\ f() \ \rightarrow\ (\xi,\ E):\ \{x_1,\ldots,x_n\},\ n > 0 \end{array}}{(\xi,\ p) \xrightarrow{//} (\xi,\ p\begin{bmatrix} state/expand \\ sons/\{(x_1,\pi),\ldots,(x_n,\pi)\} \end{bmatrix})}$$

Rule 3: expanding a node: generating the labels of the branches

$$\frac{\begin{array}{l} C = Choice(f,g,h,k) \\ p : \wp\begin{pmatrix} state = start \\ env = E \\ code = C \times T_1 \Diamond T_2 \end{pmatrix},\ (\xi,\ E):\ f() \ \rightarrow\ (\xi,\ E):\ \{\} \end{array}}{(\xi,\ p) \xrightarrow{//} (\xi,\ p[code/T_2])}$$

Rule 4: generating an empty set of labels: fixpoint state

$$\frac{\begin{array}{r} (\xi,\ E):\ g(x_i)\ \rightarrow\ (\xi,\ E'):\ void \\ (\xi,\ E):\ k()\ \rightarrow\ (\xi,\ E):\ true \\ p : \wp\begin{pmatrix} state = expand \\ env = E \\ code = Choice(f,g,h,k) \times T_1 \Diamond T_2 \\ sons = \{(x_1,p_1),\ldots,(x_i,\pi),..,(x_n,p_n)\} \end{pmatrix} \qquad p' : \wp\begin{pmatrix} father = p \\ env = E' \\ code = T_1 \\ state = start \end{pmatrix} \end{array}}{(\xi,\ p) \xrightarrow{//} (\xi,\ p[sons/\{(x_1,p_1),\ldots,(x_i,p'),..,(x_n,p_n)\}])}$$

Rule 5: going down a branch

$$\frac{\begin{array}{ll} p' : \wp\begin{pmatrix} father = p \\ state = failed \end{pmatrix} & \begin{array}{r} (\xi,\ E):\ g(x_i)\ \rightarrow\ (\xi,\ E'):\ v \\ (\xi,\ E):\ k()\ \rightarrow\ (\xi,\ E):\ v' \\ (v = \perp \vee\ v' = false) \end{array} \\ p : \wp\begin{pmatrix} state = expand \\ env = E \\ code = Choice(f,g,h,k) \times T_1 \Diamond T_2 \\ sons = \{(x_1,p_1),\ldots,(x_i,\pi),..,(x_n,p_n)\} \end{pmatrix} & (\xi,\ E):\ h(x_i)\ \rightarrow\ (\xi',\ E''):\ void \end{array}}{(\xi,\ p) \xrightarrow{//} (\xi',\ p\begin{bmatrix} env/E'' \\ sons/\{(x_1,p_1),\ldots,(x_i,p'),..,(x_n,p_n)\} \end{bmatrix})}$$

Rule 6: failure in a branch

$$\frac{p : \wp\begin{pmatrix} state = expand \\ sons = \{(x_1,p_1),\ldots,(x_n,p_n)\} \end{pmatrix},\quad \forall i,\ x_i \neq \pi}{(\xi,\ p) \xrightarrow{//} (\xi,\ p[state/wait])}$$

Rule 7: when all sons of a node have been expanded

$$\frac{\begin{array}{l}\forall i,\ p_i.state = failed \\ p : \wp\begin{pmatrix} state = wait \\ sons = \{(x_1, p_1), \ldots, (x_n, p_n)\}\end{pmatrix}\end{array}}{(\xi,\ p, p_1, \ldots, p_n, p_1', \ldots, p_l') \xrightarrow{//} (\xi',\ \{p[state/failed], p_1', \ldots, p_l'\})}$$

Rule 8: when all sons of a node report a failure

$$\frac{\begin{array}{l}\forall i\ \in \{2..n\},\ p_i.state \in \{failed,\ done\} \\ p_1.state = done \\ p : \wp\begin{pmatrix} state = wait \\ sons = \{(x_1, p_1), \ldots, (x_n, p_n)\}\end{pmatrix}\end{array}}{(\xi,\ p, p_1, \ldots, p_n, p_1', \ldots, p_l') \xrightarrow{//} (\xi',\ \{p[state/done], p_1', \ldots, p_l'\})}$$

Rule 9: when all sons of a node have finished their computations

$$\frac{p_1.code = exit}{(\xi,\ \{p_1, \ldots, p_n\}) \xrightarrow{//} (\xi',\ \{p_1[state/done]\})}$$

Rule 10: exit terminator

$$\frac{p_1.code = cont}{(\xi,\ \{p_1, \ldots, p_n\}) \xrightarrow{//} (\xi',\ \{p_2, \ldots, p_n\})}$$

Rule 11: cont terminator

$$\frac{\begin{array}{ll} p' : \wp\begin{pmatrix} state = start \\ env = E \\ code = T \end{pmatrix} & \begin{array}{r}(\xi,\ p) \xrightarrow{//} (\xi',\ \{p_1, \ldots, p_m, p_1', \ldots, p_l'\},\ m > 0 \\ \forall i \in \{1..m\},\ (p_i.state = start,\ p_i.code = \varepsilon)\end{array} \\ p : \wp\begin{pmatrix} state = start \\ env = E \\ code = T / n \times T_1 \lozenge T_2 \end{pmatrix} & \begin{pmatrix} m = n \\ \vee \begin{pmatrix} m < n \\ \forall j \in \{1..l\},\ \begin{pmatrix} p_j'.state \in \{failed, done, wait\} \\ p_j'.code \neq \varepsilon \end{pmatrix}\end{pmatrix}\end{pmatrix}\end{array}}{(\xi,\ p) \xrightarrow{//} (\xi',\ \{p_1[code/T_1], \ldots, p_m[code/T_1]\})}$$

Rule 12: limiting the number of exits of a search

$$
\frac{
\begin{array}{ll}
p' : \wp\begin{pmatrix} state = start \\ env = E \\ code = T \end{pmatrix} &
\begin{array}{r}
(\xi,\ p) \xrightarrow{//} (\xi',\ \{p_1,\ldots,p_m,p_1',\ldots,p_l'\},\ m > \\
\forall i \in \{1..m\},\ (p_i.state = start,\ \ p_i.code = \varepsilon \\
\forall j \in \{1..l\},\ (p_i.state \in \{wait, failed, done\}
\end{array} \\
p : \wp\begin{pmatrix} state = start \\ env = E \\ code = smallest(T,f,n,r) \times T_1 \Diamond T_2 \end{pmatrix} &
\begin{array}{r}
\forall i \in \{1..m\},\ (\xi,\ p_i.env) : f()\ \rightarrow (\xi,\ p_i.env) : a \\
a_1 \le a_2 \le \ldots \le a_n \\
p = \min(i \in \{1..m\}\ \mid\ a_i > a_1 / r \\
k = \min(m,n,p
\end{array}
\end{array}
}{
(\xi,\ p) \xrightarrow{//} (\xi',\ \{p_1[code/T_1],\ldots,p_m[code/T_1]\})
}
$$

Rule 13: look-ahead search

$$
\frac{
\begin{array}{ll}
p' : \wp\begin{pmatrix} state = start \\ env = E \\ code = T \end{pmatrix} & (\xi,\ p) \xrightarrow{//} (\xi',\ \{p''\}) \\
\begin{pmatrix} C = T / n \\ \vee\ \ C = smallest(T,f,n,r) \end{pmatrix} & \left(p'.state = failed \Rightarrow \begin{pmatrix} s = failed \\ c = \varepsilon \end{pmatrix} \right) \\
p : \wp\begin{pmatrix} state = start \\ env = E \\ code = C \times T_1 \Diamond T_2 \end{pmatrix} & \left(p'.state = done \Rightarrow \begin{pmatrix} s = start \\ c = T_1 \end{pmatrix} \right)
\end{array}
}{
(\xi,\ p) \xrightarrow{//} (\xi',\ \left\{ p''\begin{bmatrix} code/c \\ state/s \end{bmatrix} \right\})
}
$$

Rule 14: no exits from a look-ahead or limited search

5. Examples

5.1 Resolution strategies for a finite domain constraint solver

As a first simple example for SaLSA we express some standard labeling strategies offered as built-in in most finite domain constraint solvers. One can for instance consider three branching mechanisms :

- Selecting an unassigned variable (for example, according to the first-fail principle, picking a variable with minimal domain cardinal), and assigning successively all possible values to it. This can be described by the following choice point *(L)* :

```
L :: Choice
moves post(FD, x == v)
  with x = some(x in {x in Variable | unknown?(value,x)} minimizing card(x)),
       v ∈ values(x)
```

- For variables with many values in their domain, one usually replaces the assignment procedure by a domain reduction procedure. This can be described by this choice point *(R) :*

```
R :: Choice
moves post(FD, x <= mid) or post(FD, x > mid)
  with x = some(x in Variable | unknown?(value,x)),
        mid = (x.sup + x.inf) / 2
```

- A last procedure consists in picking a suspended disjunction (c_1 or c_2) and enforcing one the two constraints (c_i). This can be described by the choice point (*D) :*

```
D :: Choice
moves post(FD, d.c1)  or post(FD, d.c2)
  with d = some(d in Disjunctions | unknown?(side,d))
```

With all these choice points, a standard strategy for solving satisfiability problems can be expressed as :

SOLVE(D ◊ R* ◊ L* ◊ exit)*

In order to address optimization (say minimization) problems, the solver also needs to offer a branch&bound search algorithm. This can be described in the following manner. Let *UB* be a global variable storing the best value found so far in the search (initialized with a high constant), and let *OBJ* be a domain variable representing the objective function to minimize, let us defining the following utility functions.

registerUB() *-> UB := OBJ.value*
enforceUB() *->post(FD, OBJ < UB)*

The branch and bound minimizing algorithm can then be expressed with the following expression :

*SOLVE((D *◊ R* ◊ L*) where enforceUB ◊ registerUB ◊ cont)*

5.2 The *n* queens problem.

Let us now consider the famous *n*-queens problem (placing *n* queens on a *n*×*n* chessboard such that no pair of queens attack each other's position). This problem could be solved very simply with a constraint solver using the procedures described in the previous paragraph. However, for the sake of the argument, let us present a specialized search algorithm.

We build a solution to the *n* queens problem by filling up two arrays : *line* and *column*, with the following meaning : *line[x] = y* (or *column[y] = x*) means that a queen has been placed on the chessboard, in the (x,y) spot. Otherwise, line[x] = 0 (resp column[y] = 0) means that no queen has yet been placed in the line *x* (resp. in the column *y).* In order to keep both array consistent, and to perform some propagation, we use a small solver, called *chessboard* (for example, posting the update *line[x] := y* to *chessboard* triggers the update *column[y] := x*). Moreover, we compute a matrix (*possible)* of Booleans indicating which spots are not attacked by a queen already placed on the board. The *chessboard* solver is responsible for keeping this matrix up to date.

A first choice point (*L)* picks a line containing no queen and recursively tries to place a queen in all free spots in this line.

```
L :: Choice
  move post(chessboard, line[i] := j)
    with i = some(i in Lines | line[i] = 0),
          j ∈ {j in Columns | possible[i,j]}
```

From this choice point several algorithms can be tested : searching for the first solution, printing all solutions, or printing the first three solutions found.

SOLVE(L ◊ exit), SOLVE(L* ◊ display ◊ cont),*

SOLVE(L / 3 ◊ display ◊ cont)*

However, these algorithms may return solutions having less than *n* queens on the board. Since exactly one queen is placed in a branch of a choice point, boards with *n* queens will be obtained at depth *n,* whereas solutions obtained at a shallower depth correspond to less queens. In order to print only solutions with *n* queens, on can run the following algorithm.

$$SOLVE(((L^n \lozenge cont) / 3) \times display \times exit)$$

For a more complex algorithm with the *n* queens problems, one could very well reason on columns instead of lines. In fact, a more appropriate algorithm would reason sometimes on lines, sometimes on columns, depending on which model is most constrained. This can be described with the following choice point :

```
Mix :: Choice
  case Size(domain(line[i0])) < Size(domain(column[j0]))
    ({true} moves post(chessboard, line[i0] := j)
           with j ∊ {j in Columns | possible[i0,j]}
     {false} moves post(chessboard, line[i] := j0)
           with i ∊ {i in Lines | possible[i,j0]} )
  with  i0 = some(i in {i in Lines | line[i] = 0} minimizing Size(domain(i)),
        j0 = some(j in {j in Columns | column[j] = 0} minimizing Size(domain(j))
```

With this composite choice point, a simple algorithm for finding an assignment of queens to places can be described as

$$SOLVE(\ Mix^* \lozenge exit)$$

Finally, let us mention that a standard labeling technique for the *n* queens problems consists in trying first (within a line or a column), the spots near the middle of the sequence. The choice point *L* can be easily modified (*L'*) to take this heuristic into account :

```
L :: Choice post(chessboard, line[i] := j)
       with i = some(i in Lines | line[i] = 0),
            j ∊ {j in Columns | possible[i,j]}
       sorted by increasing abs(n / 2 - i)
```

5.3 Various insertion algorithms for routing problems

We now consider the well-known vehicle routing problem : the problem is defined a set of workers and by a set of tasks to be done at different locations. A solution is an assignment of tasks to workers as well as a route (a schedule) for each worker. Standard algorithms are insertion algorithms in which one picks one task at a time, affects it to a worker and inserts it at some place in its route. In order to describe such procedures, let us define choice points for representing this insertion mechanism. The first choice point describes how a task is inserted between two successive ones in a tour, the second choice point uses the first one to consider all routes, and retain only the best insertion for each route.

```
Insert(t:Task, r:Route) :: Choice
  moves link(x,t,r), link(t,y,r)
    with (x,y) ∊ edges(r)
```

```
OneInsert :: Choice
  moves from smallest(Insert(t,r), Length(r))
       with t = some(t in Tasks | unaffected(t)),
            r ∊ Routes
```

Simple look ahead heuristic recursively picks a task and a route and inserts the task where it fits best in that route :

SOLVE((OneInsert / 1) ◊ exit)*

The algorithm can be improved with some incremental local optimization. The idea is to re-optimize the solution after each insertion (in order to correct mistakes made by wrong insertions, and to improve the sequencing of tasks in the route where the task has just been inserted). However, during the look-ahead exploration of the branches, one can afford only a limited amount of optimization, while in the actual insertion, one has the time to perform a more thorough pass of local optimization. Let us define two local optimization procedures with the following choice points:

```
TwoOpt(t:Task) :: Choice
  moves link(x,u,r), link(y,z,r)
        with r = route(t)
              (xy, uz) ∈ SortedPairsOf(edges(r))
  such that delta(Length(r)) < 0
```

```
ThreeOpt(t:Task) :: Choice
  moves link(x,u,r1), link(z,y,r2), link(t,w,r)
        with r = route(t), u = next(t),
              r1 ∈ RoutesCloseTo(r), r2 ∈ RoutesCloseTo(r),
              xy ∈ edges(r1), zw ∈ edges(r2)
  such that delta(TotalLength) < 0
```

```
SmartInsert:: Choice
  moves from smallest(Insert(t,r) x (TwoOpt(t)/1)*, Length(r))
                       x (ThreeOpt(t)/1)*
        with t = some(t in Tasks | unaffected(t)),
```

SOLVE((SmartInsert/1) ◊ exit)*

5.4 Various Scheduling heuristics.

We now describe a set of search algorithms dedicated to scheduling problems. This first ones are constructive heuristic based on the notion of selection. Tasks are selected one by one and scheduled as soon as possible accordingly with their resource requirements. This process can be described with the choice point :

```
Ins :: Choice
  moves post(schedule,t.start = t.start.inf)
    with t ∈ {t in Tasks | available(t) & not(scheduled(t))}
```

yielding a naive heuristic :

SOLVE((Ins/1) ◊ exit)*

This heuristic can be improved if we decide to choose the task to schedule by some priority rule (for example, the shortest or the most urgent task). The algorithm then becomes :

```
Ins2 :: Choice
  moves post(schedule,t.start = t.start.inf)
      with t ∈ {t in Tasks | available(t}
  sorted by decreasing priority(t)
```

SOLVE((Ins2 / 1) ◊ exit)*

As for the previous routing examples, this priority measure can also be assessed after the insertion, yielding a look-ahead search algorithm.

$$SOLVE(largest(Ins, global\_cost, 1)^* \lozenge exit)$$

Since these enriched greedy strategies yield good results, they can be enriched with some global backtracking. This yields a kind of limited discrepancy search algorithm (see [HG 95]), where branching takes place when two tasks *t* and *t'* were candidates to be inserted, which could not be distinguished by our heuristic evaluation function. In these cases, instead of breaking ties arbitrarily, we create a backtrack point and consider both possibilities (insert t, insert t'), in separate branches. This yields the following algorithm :

$$SOLVE(smallest(Ins, bound, 2, 1.0)^* \lozenge display \lozenge exit)$$

In order to have a finer control over the size of the search tree, one can also adapt the amount of backtracking to the depth in the tree. This is motivated by the fact that backtracking at the root of the tree is responsible for more diversification in the solution set than backtracking deeper in the tree. This mechanism can be (for example) implemented by the following algorithm :

$$\begin{aligned} SOLVE(\quad & smallest(Ins, bound, 5, 1.0)^{5} \\ \times\ & smallest(Ins, bound, 2, 1.0)^{10} \\ \times\ & smallest(Ins, bound, 1)^* \lozenge exit) \end{aligned}$$

5.5 Exhaustive scheduling techniques and "shaving"

Let us now consider an exhaustive approach to the (disjunctive) scheduling problem. Instead of selecting the tasks one by one and scheduling them as soon as possible, one may consider pairs of tasks and decide which one should be processed first. This technique, called « edge-finding », is well adapted to an exhaustive search (exact optimization) since the branches of the search tree are mutually exclusive (which was not always the case of the algorithms described in the previous paragraph). This search procedure can be described as follows :

```
EdgeFinding :: Choice
  move   post(schedule, t1 isbefore t2)
      or post(schedule, t2 isbefore t1)
    with (t1, t2) = some{pair in tuple(Task, Task) | unordered(pair)}
```

Another exhaustive search approach would be to affect to each task a starting date, by trying all possible dates, one after the other. This search algorithm is highly inefficient as an exhaustive method (because it produces many equivalent schedules). However, it may be used as a propagation mechanism through limited look-ahead search. This propagation technique is called « shaving » [MS 95] and it is based on the following idea :

Let *t* be a task with possible starting times between *a* and *b*. We try to force *t* to start at time *a*, and propagate : if this assignment leads to a failure, the domain of the starting time of *t* can be « left-shaved », i.e. its lower bound may be updated to *a+1*. This process is repeated until some *k* is reached such that assigning the *a+k* as a starting date for *t* no longer raises a contradiction (a procedure shaving domains from the right can be designed in a similar manner). A choice point expressing this labeling procedure can be described as :

```
Start(t:TASK) :: Choice
  move post(schedule,t.start = v)
    with t = some(t in Task | unknown(value, t.start)},
         v ∈ (t.start.inf .. t.start.sup)
  on failure t.start != v
```

The shaving procedure amounts to a limited visit of the *Start* choice point, which we want to stop as soon as one leaf has been reached (without causing failure), and for which we want to exit not in a leaf, but at the root node.

$$SOLVE(Start(t) / 1 \times cont)$$

5.6 Local Optimization and partial schedules

In this paragraph, we present a local optimization algorithm based on tree search. The idea is to go from one solution to one of its neighbors by keeping the good part of the initial solution and trying to re-plan the rest of it. Such a procedure differs from the usual hill-climbing control of local optimization, in the sense that the choice of the part of the solution that is being kept is non-deterministic and that several trials are allowed (this procedure is described as « generalized shuffle » in [CL 95]).

To describe such an algorithm, we need two choice points: a first one, *Forget* selects the partial solution to keep from the initial solution, a second one, *EdgeFinding* which orders two tasks sharing the same resource (this branching procedure has been presented earlier as the standard method for exhaustive search). We are now able to write a move which tries at most three times to keep only part of the current solution, and which, in each case, tries to re-obtain a full schedule by performing a global search limited by 100 backtracks.

$$SOLVE(Forget / 3 \times (EdgeFinding* \; ! \; 100) \times exit)$$

5.7 Playing chess

The next example concerns chess playing. The two players play one after the other and each one seeks the move which will put him into the best position. Let us assume that a configuration of the chess board can be assessed by a function (the better the configuration, the higher the value) f_w for the white player just after a white move (and similarly a function f_b for the black player just after a black move). Moreover, let *Wmove* be a choice point iterating all possible moves for the white player. A one-move ahead strategy for the white player can hence be described by the term

$$SOLVE(largest(Wmove, f_w) \Diamond \; exit)$$

A more sophisticated strategy could also examine all possible replies of the black player to the considered move. This is described as

$$SOLVE(smallest(Wmove, max(Bmove, f_b)) \Diamond \; exit)$$

The white player may also decide to reason three moves ahead, but considering only the five replies of the black player, which seem best. This can be described as

$$SOLVE(largest(Wmove, min(largest(Bmove, f_b, 5), max(Wmove, f_w))) \Diamond \; exit)$$

A full search tree based on choice points as the ones which have just been described would then correspond to a simulation of the chess game.

5.8 Redundant disjunctive scheduling constraints : searching for « good » maximal cliques

We now address a problem that we encountered on some instances of cumulative scheduling problems. In these instances, the standard cumulative scheduling algorithm behaved poorly because it was not able to make some simple inferences. These simple inferences could have been made if one had realized that many pairs tasks of tasks were disjunctive (i.e. could not be in process at the same time), either because their combined resource requirement exceeded the available quantity or because of explicit precedence constraints. When there are many such disjunctions, it is interesting to consider cliques in the graph of these disjunctions (a clique of disjunctions is a set of tasks such that any pair of them is disjunctive). In fact, to each clique of disjunctions, one can create a fake disjunctive resource and impose that all tasks in the clique require this resource. This is a redundant constraint and does not discard any solution, but it may perform new inferences.

In fact, the most interesting cliques are the maximal ones. However, there are too many of them and one cannot afford to add too many such redundant constraints (the slow down in execution time is noticeable). So, one has to find a set of « good » maximal cliques. For example finding the cliques with maximal cardinality or with maximal duration (adding the durations of the tasks in the clique). This can be done with a simple search algorithm, trying to build the set of tasks S corresponding to a clique :

SOLVE(largest(C, totalduration, 3, 0.90)* $\times$ *make_redundant_constraint* $\times$ *cont)*

with a simple choice point *C :*

```
C :: Choice
  moves S :add t
      with t ∊ {t in Task | forall(t′ in S, disjunctive(t,t′)) }
      sorted by decreasing duration(t)
```

5.9 The Lin & Kernighan heuristic for the TSP

We now describe a local optimization procedure for the traveling salesman problem (TSP), due to Lin and Kernighan [LK 73]. The procedure generalizes the standard *k*-opt moves, which consist in removing *k* edges from the tour and replacing them by *k* other ones. The idea is based on the following observation : a *k* opt move, always consists in replacing a set of edges $\{e_1,...,e_k\}$ by $\{f_1,...,f_k\}$. Without losing too much generality, one can assume that for all *i*, e_i and f_i have the same end point. The transformation can hence be constructed incrementally, by replacing e_1 par f_1, e_2 by f_2, etc. When the move yields an improvement, we get

$$\sum_{i=1}^{k}(d_{e_i}-d_{f_i})>0\cdot$$

The key point is then to notice that the indices *i* may be ordered in such a way that for all partial sums S_l, $S_l > 0$.

$$S_l=\sum_{i=1}^{l}(d_{e_i}-d_{f_i})\cdot$$

The LK heuristic for finding one move is the following. Starting from some vertex *x'* in the tour, we replace the incoming edge on *x'* e_1=*xx'* by a shorter one f_1 =*yx'* ($d_{f_1} < d_{e_1}$). Then, we replace the outgoing edge on *y,* e_2=*yy'* by some other f_2 =*zy'* such that $d_{f_1} + d_{f_2} < d_{e_1} + d_{e_2}$, and so on ... A limited amount of backtracking is performed to find the best move (in the original paper, backtracking occurs only on the two first levels of the search). Once the best move has been found, it is generally used in a hill-climbing walk.

Here is the full example of a program searching for a move with the L&K heuristic :

```
// data in the global environment:

      d[x:City, y:City] : integer          // the distance matrix
      pred[x:City] : City                  // the immediate predessor of a city
                                           //  (used to store the tour to optimize)
      possible_pred[x:City] : set[Cit]y    // the candidates to be the immediate predessor

// data in the local environments (in each node of the tree)

      l:list[City] := nil

// The solver LK

      LK :: invariant(gain = sum(d[pred[l[i]], l[i]] - d[l[i+1] - l[i], i in (1 .. length(l - 1)) )
```

// functions

__cyclic__() : boolean -> length(l) > 1 & l[1] = l[length[l]]

__notbetter__() : boolean ->gain ≤ 0

// choice points

```
F :: Choice
  moves post(LK, l := list(x))
          with x ∊ Cities
```

```
N :: Choice
  movespost(LK, l :add x)
    with x ∊ {x in Cities | exists(y in possible_pred[x] | d[y,x] < d[pred[x],x]]}
  such that gain ≥ 0
```

the algorithm for finding a move can then be expressed as follows :

SOLVE(largest(F $\times$ *(N* $\times$ *(N / 1)* until cyclic), gain, 1)*

and the overall local search algorithm can be described as

SOLVE(largest(F $\times$ *(N* $\times$ *(N / 1)* until cyclic), gain, 1)* until notbetter ◊ exit)*

Hence, this complex heuristic can be programmed in very simple manner with SaLSA. Moreover, the user can very simple experiment some other backtracking possibilities, such as the following algorithm which performs an exhaustive search of neighborhoods containing at most 3 moves

```
SOLVE(largest(   largest(  F × ((N / 2)^8,  × (N / 1)* until cyclic),
                               gain, 3)* until notbetter,
                 gain, 1) ◊ exit)
```

6. A first implementation

In this section, we report some specific features of the first implementation that was realized as a Claire pre-processor.

6.1 Code generation

A prototype implementation of SaLSA has been realized from the operational semantics. The implementation is mono-process and only one strategy of visit of the search tree is offered: depth first search. The current implementation is a Claire generator, which produces CLAIRE objects and methods from a SaLSA code.

The main interest of such an implementation is its memory cost. Except for the terms involving look-ahead search (with smallest/largest), all the nodes of the search tree that are under consideration at a given time instant are always on a same branch, and only the deepest one is active. Hence, the environments of all processes can be stored (incrementally) on a stack. Such mechanisms are already offered by the Claire language, which makes the generation of code easier.

For the look-ahead search procedures (say *smallest(T, f, k)*), there are two possibilities for the implementation : one can either keep a copy of all potential interesting environments, and throughout the exploration of *T,* keep only the *k* best ones. One can also remember only the paths in the search tree rather than the whole environments. This amounts to a time versus space tradeoff : the first solution is much more memory consuming but may be faster (the environments need not be reconstructed), however, never more than by a factor 2. In practice when *T* is a large tree, and *k* is small, the speed gain is likely to be negligible, whereas the memory requirements are not bounded. Therefore, we settled for the second implementation.

The current implementation is a prototype and is still under development. However, we expect to release a first system in the public domain in the fall of 1997, together with a small finite constraint domain solver, called ECLAIR.

6.2 ASCII syntax

In the implementation of SaLSA, the syntax for describing choice points is different from the one that has been used throughout this document. The first reason is due to the limitation of ascii characters : the operators $\times$, $\Diamond$ and $\in$ are replaced by `*`, `/+` and `in`, the terms C^n and C^* are written `C ^ n` and `C ^ *`. Moreover, the syntax for choice points is different : it is less readable, but the code generator was easier to implement. The difference is that we use functions instead of arbitrary Claire expressions with possibly free variables. The branching process is separated in two phases. First, generating a set of exclusive decisions, and then, taking them. In order to describe the set of decisions in a parametric manner, the decisions are represented by the evaluation of a single function on different sets of parameters (so here, we do not allow the explicit neighborhood syntax <move> or <move>, it needs to be coded into one single parametric function call). This amounts to decomposing choice points into an iteration and a function :

We start by describing the creation of the variables. Each variable is either bound to one single value (x = <exp>) or to a set of values (x $\in$ <exp>). Suppose there is only one variable. We encapsulate the expression defining its value in a function *f* (without parameters since the function had no free variables) and denote this part of the choice point by *Choose(f)* if *f* returns a set of values and by *Focus(f)* is *f* returns a single value. When there are several variables, each one of them is defined by a *Focus(<function>)* or *Choose(<function>)* construct. However, the variables are not always independent one from the other (consider for example a simple labeling procedure featuring two variables: an unassigned domain variable (one single object) and a set of possible values from its domain (a set of objects : Here, the variable needs to be generated before the set of values since the values depend on the variable.

To express these dependencies between variables, the *Focus* and *Choose* constructs will be composed with the 'o' operator, in sequence. For example, the following expression

***Choose**(g)* ***o Focus**(f)* ***o Choose**(h)*

yields as parameters the set of triples *(x,y,z)* such that :

$x \in g(),\ y = f(x),\ z \in h(x,y)$

The arguments *x* is always passed as parameter to the function *f*, so the function defining the i^{th} variable must always be of arity (i-1) (even if it doesn't use all its parameters). This allows us to use all possible free variables that we may want. Note that passing functions is strictly equivalent to passing expressions when one declares the function as an *inline* function. Once this set of parameters is constructed (which is can also naturally be done incrementally), the label generator can be enriched with a function to evaluate on these parameters in order to take the decision. The addition of the decision function is done through the use of the *do* operator. For example, the choice point

```
Label :: Choice
  moves assign(x,v)
    with x = NextVariable()
         v ∈ values(x)
```

can be entered as followed on the system

Focus**(NextVariable)* ***o Choose**(Values)* ***do *assign*

The side effects of the evaluation of the *assign* function are undone upon backtracking to the father node. However, one can also perform some actions in the father node when one of its children reports a failed search. In order to do so, one can apply the *bk* operator to a choice point, in the same manner as the *do* operator. For example, the choice point

```
Label2 :: Choice
  moves assign(x,v)
    with x = NextVariable()
         v  ∊ values(x)
    on failure updateGUI(x,v)
```

can be entered as followed on the system

Focus*(NextVariable)* ***o Choose****(Values)* ***do*** *assign* ***bk*** *updateGUI*

7. Conclusions

In this paper, we have presented a language for describing specialized (global and local) search algorithms. This allows us to specify in a very concise manner the flow of control for complex algorithms. Such a language is a step towards programming complex hybrid algorithms with a high level of abstraction. By raising the level of abstraction, the code becomes shorter and more elegant (yielding shorter development and maintenance times). By using a code generator, we can improve the reliability of the run-time code. Future directions of research include a programming environment dedicated to such algorithms (with a special purpose debugger, stepper, tracer, etc.)

8. Acknowledgments

Part of this work (including the first implementation prototype) was done while François was working at the Corporate Research Laboratories of Thomson-CSF. We would like to thank Jean Jourdan, the ATS laboratory and Thomson-CSF for supporting this work. We would also like to thank the teams at Bouygues, at the LIENS and partners of the CHIC-2 project for many fruitful discussions. We are specially grateful to François Fages, Eric Jacopin, Laurent Perron and Mark Wallace.

9. References

[AS 97] K. Apt, A. Schaerf, *Search and Imperative Programming,* Proceedings of POPL' 97, ACM, 1997.

[CL 95] Y. Caseau F. Laburthe*, Disjunctive Scheduling with Task Intervals*, LIENS Technical Report 95-25, École Normale Supérieure, 1995.

[CLAIRE] Y. Caseau F. Laburthe*, Introduction to the CLAIRE programming language*, LIENS Technical Report 96-15, École Normale Supérieure, 1996.

[LK 73] Lin, Kernighan

[Mei 95] M. Meier *GRACE: The Graphical Constraint Environment*, 1996.

[MS 95] P. Martin, D. Shmoys, *A New Approach to Computing Optimal Schedules for the Job Shop Scheduling Problem*, in M. Queyranne (ed): Proceedings of IPCO'5, LNCS, Springer, 1996.

[Sch 97] C. Schulte, *Oz Explorer: A visual constraint programming tool*, Proceedings of ICLP'97, Lee Naish ed., The MIT Press, 1997.

[SS 94] C. Schulte, G. Smolka, *Encapsulated Search for Higher-Order Concurrent Constraint Programming*, Logic Programming, Proceedings of ILPS'94, The MIT Press, 1994

[MvH 97] L. Michel, P. van Hentenryck, *Localizer: A modeling language for local search* to appear in Proc. of Constraint Programming CP'97, LNCS, Springer, 1997

[Zi 93] S. Zilberstein *Operational Rationality through Compilation of Anytime Algorithms*, Ph. D. dissertation, Computer Science division, University of California at Berkeley, 1993.