Nondeterministic Lisp as a Substrate for Constraint Logic Programming

Jeffrey Mark Siskind[1]
University of Pennsylvania
Institute for Research in Cognitive Science
3401 Walnut Street Room 411C
Philadelphia PA 19104
215/898–0367
internet: Qobi@CIS.UPenn.EDU

David Allen McAllester[2]
M. I. T. Artificial Intelligence Laboratory
545 Technology Square
Cambridge MA 02139
617/253–6599
internet: dam@AI.MIT.EDU

Abstract

We have implemented a comprehensive constraint-based programming language as an extension to COMMON LISP. This constraint package provides a unified framework for solving both numeric and non-numeric systems of constraints using a combination of local propagation techniques including binding propagation, Boolean constraint propagation, generalized forward checking, propagation of bounds, and unification. The backtracking facility of the nondeterministic dialect COMMON LISP used to implement this constraint package acts as a general fallback constraint solving method mitigating the incompleteness of local propagation. An important aspect of our implementation is that it is both an extension to COMMON LISP as well as fully implemented within COMMON LISP. This is in contrast to recent constraint-based languages formulated in the logic programming framework which cannot be implemented efficiently in the language which they extend.

CONTENT AREA: Constraint-Based Reasoning

[1]Supported in part by an AT&T Bell Laboratories Ph.D. scholarship to the author, by a Presidential Young Investigator Award to Professor Robert C. Berwick under National Science Foundation Grant DCR–85552543, by a grant from the Siemens Corporation, and by the Kapor Family Foundation. Also supported in part by ARO grant DAAL 03–89–C–0031, by DARPA grant N00014–90–J–1863, by NSF grant IRI 90–16592, and by Ben Franklin grant 91S.3078C–1

[2]Supported in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-91-J-4038.

1 Introduction

Recent years have seen significant interest in constraint logic programming languages. Numerous implementations of such languages have been described in the literature, notably CLP($\Re$) (Jaffar and Lassez 1987) and CHIP (Van Hentenryck 1989). The point of departure leading to these systems is the observation that the unification operation at the core of logic programming can be viewed as a method for solving equational constraints between logic variables which range over the universe of Herbrand terms. A natural extension of such a view is to allow variables to range over other domains and augment the programming language to support the formulation and solution of systems of constraints appropriate to these new domains. The notion of extending a programming language to support constraint-based programming need not be unique to logic programming. In this paper we present the constraint package included with SCREAMER, a nondeterministic dialect of COMMON LISP described in a companion paper. This package provides functionality analogous to CLP($\Re$) and CHIP in a COMMON LISP framework instead of a PROLOG one. We argue that for a number of reasons, nondeterministic LISP offers a better substrate for implementing and using constraint-based programming techniques than PROLOG does.

SCREAMER is our implementation of nondeterministic COMMON LISP. It is described in a companion paper (Siskind and McAllester 1993). SCREAMER augments COMMON LISP with the capacity for writing nondeterministic functions and expressions. Nondeterministic functions and expressions can return multiple values via backtracking. Only two primitives must be added to COMMON LISP to support nondeterminism. The first is the macro `(either` e_1 e_2`)` which first evaluates and returns the value of e_1, and later upon backtracking evaluates and returns the value of e_2. The second is the function `fail` which initiates backtracking. Using these two primitives one can construct a function which nondeterministically returns the members of a set one at a time via backtracking.

```
(defun a-member-of (x)
 (if (null x) (fail))
 (either (first x) (a-member-of (rest x))))
```

One can then adopt a generate-and-test style of programming to solve constraint satisfaction problems.

```
(let ((x1 (a-member-of s1)) ... (xn (a-member-of sn)))
 (unless Φ[x1...xn] (fail))
 (list x1...xn))
```

The bulk of this paper will present programming constructs which support more efficient techniques for solving the same class of problems.

In addition to `either` and `fail`, SCREAMER provides a number of other primitives which are less central to our current concerns. One primitive which is crucial, however, is the ability to perform *local* side effects, ones which are undone upon backtracking. Local side effects are specified by enclosing a `setf` expression inside a `local` declaration, e.g. `(local (setf` v e`))`. Unlike logic variables in PROLOG which allow only single assignment local side effects through the unification binding operation, local side effects in SCREAMER may repeatedly change the value of a variable and still have such changes undone in reverse order at the appropriate times during backtracking. Nondeterminism and local side effects form the substrate on top of which the SCREAMER constraint package is constructed.

SCREAMER is an extension of the COMMON LISP programming language which provides both the capability for writing nondeterministic programs as well as the constraint package described in this paper. One novel aspect of SCREAMER is that it is itself written in COMMON LISP and thus is fully portable to any complete COMMON LISP implementation. The lowest level of SCREAMER implements the nondeterminism facility. The constraint package is then built on top of this substrate.

2 Variables and Constraints

SCREAMER includes the function `make-variable` which returns a data structure called a *variable*. SCREAMER variables are a generalization of PROLOG logic variables. Initially, new variables are unbound and unconstrained. Variables may be bound to values by the process of solving constraints asserted between sets of variables. Binding is done with local side effects. Thus variables become unbound again upon backtracking.

SCREAMER provides a variety of primitives for constraining variables. Each constraint primitive is a "constraint version" of a corresponding COMMON LISP primitive. For example, the constraint primitive `+v` is a constraint version of `+`. An expression of the form `(+v` x y`)` returns a new variable z, which it constrains to be the sum of x and y by adding the constraint $z =$ `(+` x y`)`. By convention, a SCREAMER primitive ending in the letter `v` is a constraint version of a corresponding COMMON LISP primitive. Table 1 lists the constraint primitives provided by SCREAMER. All of these primitives have the property that they accept variables as arguments—in addition to ground values—and return a variable as their result. The constraint primitive installs a constrain between the arguments and the returned variable stating that, under any interpretation of the variables involved, the value of the result variable must equal the corresponding COMMON LISP primitive applied to the values of the arguments. As another example, the expression `(<v` x y`)` returns a variable z and adds the constraint $z =$ `(<` x y`)`. This constraint is satisfied when z is either `t` or `nil` depending on whether x is less than y. For the most part, each constraint primitive obeys the same calling convention as the corresponding COMMON LISP primitive. SCREAMER performs a variety of optimizations to improve run time efficiency. In particular, if the value of a variable returned by a constraint primitive can be determined

Type Restrictions:	`numberpv realpv integerpv booleanpv memberv`
Boolean:	`andv orv notv`
Numeric:	`<v <=v >v >=v =v /=v +v -v *v /v minv maxv`
Expression:	`equalv`
Functions:	`funcallv applyv`

Table 1: The constraint primitives provided by Screamer.

at the time the function is called then that value is returned directly without creating a new variable.

In Screamer, most constraints are of the form $z =$ `(f` x_1 `...` x_n`)` where `f` is the Common Lisp primitive corresponding to some constraint primitive. Constraints of this form can imply type restrictions on the variables involved. For example, a constraint of the form $z =$ `(<` x y`)` implies that z is "Boolean", i.e., either `t` or `nil`. Furthermore, this constraint implies that x and y are numeric. In practice, a variable usually has a well defined type, e.g., it is known to be Boolean, known be real, known to be a cons cell, etc. Knowledge about the type of a variable has significant efficiency ramification in Screamer's constraint satisfaction algorithms. Screamer has special procedures for inferring the types of variables. Because knowledge of the types of variables is important for efficiency, in contrast to the Common Lisp primitives `and`, `or`, and `not` which accept any arguments of any type, the Boolean constraint primitives `andv`, `orv`, and `notv`) require their arguments be Boolean. This allows the use Boolean constraint satisfaction techniques for any constraints introduced by Boolean primitives. Similarly, constraint "predicates" return Boolean variables. For example, in contrast to the Common Lisp primitive `member` which can return the sub-list of the second argument whose head satisfies the equality check, the variable returned as the value of the expression `(memberv` x `'(a b c d))` is constrained to be Boolean.

Screamer includes the primitive `assert!` which can be used to add constraints other than those added by the constraint primitives. Evaluating the expression `(assert!` x`)` installs the constraint that x equals `t`. This can be used in conjunction with other constraint primitives to install a wide variety of constraints. For example, `(assert! (<v` x y`))` effectively installs the constraint that x must be less than y.[3] Certain constraint primitives in table 1, in conjunction with `assert!`, can be used to directly constrain the type of a variable. For example, evaluating `(assert! (numberpv` x`))` effectively installs the

[3]To mitigate the apparent inefficiency of this conceptually clean language design, the implementation optimizes most calls to `assert!`, such as the calls `(assert! (notv (realpv` x`)))` and `(assert! (<=v` x y`))`, to eliminate the creation of the intermediate Boolean variable(s) and the resulting local propagation.

constraint that x must be a number. Likewise evaluating `(assert! (booleanpv` x`))` installs the constraint that x must be Boolean. This is effectively the same as evaluating `(assert! (memberv` x `'(t nil)))`.

All constraints in SCREAMER are either installed by `assert!` or by one of the constraint primitives in table 1. A constraint installed by `assert!` states that a certain variable must have the value `t`. A constraint installed by a constraint primitive always has the form $z =$ `(f` x_1 ... x_n`)` where `f` is either a COMMON LISP primitive or a slight variation on a COMMON LISP primitive. The variations arise for constraint primitives such as `orv` and `memberv` where the semantics of the constraint version differs slightly from the semantics of the corresponding COMMON LISP primitive as discussed above.

An attempt to add a constraint fails if SCREAMER determines that the resulting set of constraints would be unsatisfiable. For example, after evaluating `(assert! (<v` x `0))` a subsequent evaluation of `(assert! (>v` x `0))` will fail. A call to a constraint can fail when it would generate a constraint inconsistent with known type information. For example, if x is known to be Boolean then an evaluation of `(+v` x y`)` will fail.

3 Constraint Propagation

In this section we discuss five kinds of constraint propagation inference processes performed by SCREAMER. First, SCREAMER implements binding propagation, an incomplete inference technique sometime called value propagation. Second, SCREAMER implements Boolean constraint propagation (BCP). This is an incomplete form of Boolean inference that can be viewed as a form of unit resolution (McAllester 1990). Third, SCREAMER implements generalized forward checking (GFC). This is a constraint propagation technique for discrete constraints used in Van Hentenryck's (1989) CHIP system. Fourth, SCREAMER implements bounds propagation on numeric variables. This bounds propagation—when combined with the range splitting technique discussed in section 4—implements a version of the interval method of solving systems of nonlinear equations (Hansen 1968, Hansen and Walster 1990). Finally, SCREAMER implements unification. Unification is viewed as constraint propagation inference technique which can be applied to equational constraints involving variables that range over expressions. The constraint propagation techniques are incrementally run to completion whenever a new constraint is installed by `assert!` or one of the constraint primitives. The five forms of constraint propagation are described in more detail below.

Each form of constraint propagation can be viewed as an inference process which locally derives information about variables. All forms of propagation are capable of inferring values for variables. For example, after evaluating `(assert! (orv` x y`))` and `(assert! (notv` x`))` BCP will infer that y must have the value `t`. If some constraint propagation inference process has determined a value for some variable x then we say that x is *bound* and the inferred value of x is called the *binding* of x.

Binding Propagation As noted above, most constraints in SCREAMER are of the form $z =$ `(f` x_1 `...` x_n`)` where f is a COMMON LISP primitive, z is a variable, and each x_i is either a variable or a specific value. For any such constraint SCREAMER implements a certain value propagation process. More specifically, if bindings have been determined for all but one of the variables in the constraint, and a binding for the remaining variable follows from the constraint and existing bindings, then this additional binding is inferred. This general principle is called *binding propagation*. Binding propagation will always bind the output variable of a constraint primitive whenever the input variables become bound. For example, given the constraint $z =$ `(+` x y`)`, if x is bound to 2 and y is bound to 3, then binding propagation will bind z to 5. Often, however, binding propagation will derive a binding for an input from a binding for the output. For example, given the constraint $z =$ `(+` x y`)`, if z is bound to 5 and x is bound to 2, then binding propagation will bind y to 3.

Boolean Constraint Propagation Boolean constraint propagation (BCP) is simply arc consistency (cf. Mackworth 1992) relative to the Boolean constraint primitives `andv`, `orv`, and `notv`. BCP, like arc consistency, is semantically incomplete. For example, after evaluating `(assert! (orv` z w`))` and `(assert! (orv (notv` z`)` w`))`, any variable interpretation satisfying the installed constraints must assign w the value `t`. However, BCP will not make this inference. Semantic incompleteness is necessary in order to ensure that the constraint propagation process terminates quickly. In section 4 we discuss the general fallback that we provide to backtracking search interleaved with constraint propagation to mitigate the incompleteness of local propagation.

Generalized Forward Checking Generalized forward checking (GFC) applies to variables for which a finite set of possible values has been established. Such variables are called *discrete*. For example, after evaluating `(assert! (memberv` x `'(a b c d)))` the variable x is discrete because its value is known to be either `a`, `b`, `c`, or `d`. Boolean variables are a special case of discrete variables where the set of possible values contains `t` and `nil`. Similarly, bounded integer variables are considered to be discrete. For each discrete variable SCREAMER maintains a list of possible values. The list of possible values for a given variable can be updated by the GFC inference process. The GFC inference process operates on constraints of the form $z =$ `(funcall` f x_1 `...` x_n`)`. These constraints are generated by the constraint primitive `funcallv`. Unlike most constraint primitives, the primitive `funcallv` will signal an error—rather than fail— if its first argument is bound to anything but a deterministic procedure. Now consider the constraint $z =$ `(funcall` f x_1 `...` x_n`)`. GFC will only operate on this constraint when f is bound and all but one of the remaining variables in the constraint have been bound. If the unbound variable is the output variable z, then GFC simply derives a binding for z by applying f. If the unbound variable is one of the arguments x_i then GFC tests each possible value v of the discrete variable x_i for consistency relative to this constraint. Any possible value of x_i not consistent with the constraint is removed from the set of possible values of x_i. For example, suppose that we have evaluated `(assert! (memberv` x `'(1 5 9)))`, `(assert! (memberv` y `'(3 7 12)))` and `(assert! (funcallv #'<` x y`))`. In this case the output variable of the `funcallv` constraint is bound to `t`. Now suppose that some constraint propagation inference process infers that y is 3. In this case GFC will run on the `funcallv` constraint and remove 5 and 9 from the set of possible values of x. Whenever there is only a single possible value remaining for a discrete variable, the system binds the remaining variable to that value. An example of GFC running on the N-Queens problem is given in section 5.

Bounds Propagation Bounds propagation applies to numeric variables. For each numeric value the system maintains an upper and lower bound on the possible values of that variable. These bounds propagate through constraints generated by numeric constraint primitives such as `+v`, `*v` and `<v`. For example, after evaluating `assert! (=v` x `(+v` y z`))`, if x is known to be no larger than 5.7, and y is known to be no larger than 2.2, then bounds propagation will infer that z is no larger than 3.5. Bounds propagation can also derive values for the Boolean output variables of numeric constraint predicates such as `<v` and `=v`. For example, if we have the constraint $z =$ `(<` x y`)` and the system has determined that x is at least 2.0 but y is no larger than 1.0, then the system will infer than z is `nil`.

Bounds propagation will not infer a new bound unless the new bound reduces the known interval of the variable involved by at least certain minimum percentage. This ensures that the bounds propagation process terminates fairly quickly. For example, SCREAMER avoids the very large number of bounds updates that would result from the constraints `(assert! (>v` x `0))`, `(assert! (<v` x `1000))` and `(assert! (<v` x `(-v` x `0.001)))`.

Unification Unification operates on constraints of the form $w =$ `(equal` u v`)` which are generated by the constraint primitive `equalv`. At any given time there is a system of equations defined by the set of `equalv` constraints whose output variable has been bound to `t`. SCREAMER incrementally maintains a most general unifier σ for this system of equations. For example, evaluating `(assert! (equalv (cons` x x`) (cons` y w`)))` will result in a unifier σ that equates x, y, and w, i.e., a unifier σ such that $\sigma[x]$, $\sigma[y]$, and $\sigma[w]$ are all equal.

The different forms of constraint propagation can interact with each other. For example, a given variable can be both discrete and numeric. If a discrete variable is numeric then any non-numeric possible value is eliminated. If a bound is known for a numeric variable that is also discrete then any possible value violating that bound is eliminated. SCREAMER also derives bounds information from the set of possible values of a discrete numeric variable. Unification also interacts with SCREAMER bindings. For example, if σ is the most general unifier maintained by SCREAMER, and x and y are two variables such that $\sigma[x]$ equals $\sigma[y]$, then any binding for x becomes a binding for y and vice versa. If $\sigma[x]$ equals $\sigma[y]$, and x and y have incompatible bindings, then a failure is generated.

4 Solving Systems of Constraints

By design, all of the constraint primitives described so far use only fast local propagation techniques. Such techniques are necessarily incomplete; they cannot always solve systems of constraints or determine that they are unsolvable. SCREAMER provides a number of primitives for augmenting local propagation with backtracking search to provide a general mechanism for solving systems of constraints. One such primitive, `linear-force`, can be applied to a variable to cause it to nondeterministically take on one of the values in its domain. `Linear-force` can be applied only to a variable that has a finite set of allowed values (i.e. one constrained by `memberv`) or to an integer variable. Constraining a variable to take on a value using `linear-force` may cause local propagation. Thus a single call to `linear-force` may cause a number of variables to be bound, or alternatively may fail if the variable cannot consistently take on any value. A second primitive, `divide-and-conquer-force`, can be applied to a variable to nondeterministically reduce the set of possible values it may take on. `Divide-and-conquer-force` can be applied either to a variable that has a finite set of allowed values or a real variable with finite upper and lower bounds. In the first case, the set of allowed values is split into two subsets and the variable nondeterministically constrained to take on values from either the first or second subset. In the second case, the interval is split in half and the variable nondeterministically constrained to take on values in either of the two subintervals.

The above two functions operate on single variables. More generally, one must find the values of several variables which satisfy the given constraints. SCREAMER provides two primitives to accomplish this. Both are higher-order functions which take a single variable force function as an argument (e.g. `linear-force` or `divide-and-conquer-force`) and produce a function capable of forcing a list of variables using that force function. Each incorporates a different strategy for choosing which variable to force next. The first, `static-ordering`, simply forces the variables in the order given. The single variable force function is repeatedly applied to each variable, until that variable takes on a ground value, before proceeding with the next variable. All variables are bound upon termination. The second, `reorder`, selects the variable with the smallest domain, applies the single variable force function to this variable, and repeats this process until all variables are bound. Since the choice of single variable force function is orthogonal to the choice of variable ordering strategy, SCREAMER thus provides four distinct constraint solving strategies. More can easily be added.

5 Examples

We will illustrate the power of the SCREAMER constraint language with three small examples. The first, shown in figure 1, solves the N-Queens problem. The function `n-queensv`

```
(defun attacks? (qi qj distance)
 (or (= qi qj) (= (abs (- qi qj)) distance)))

(defun n-queensv (n)
 (solution
  (let ((q (make-array n)))
   (dotimes (i n) (setf (aref q i) (an-integer-betweenv 1 n)))
   (dotimes (i n)
    (dotimes (j n)
     (if (> j i)
         (assert!
          (notv (funcallv #'attacks? (aref q i) (aref q j) (- j i)))))))
   (coerce q 'list))
  (reorder #'domain-size
           #'(lambda (x) (declare (ignore x)) nil)
           #'<
           #'linear-force)))
```

Figure 1: A constraint-based SCREAMER program for solving the N-Queens problem.

creates a variable for each row and constrains each row variable to take on an integer between 1 and n indicating the column occupied by a queen in that row. The function (a-member-ofv s) is simply syntactic sugar for:

```
(let ((v (make-variable))) (assert! (memberv v s)) v)
```

The SCREAMER primitive (solution x f) gathers all of the variables nested inside the structure x, applies the multiple variable forcing function f to this list of variables, and returns a copy of x where the variables have been replaced by their bound values.

In the above example, SCREAMER applies GFC as the technique for solving the underlying constraint satisfaction problem. SCREAMER chooses this technique since all of the variables involved have finite enumerated domains.

The second example, shown in figure 2, illustrates how bounds propagation can be used to solve systems of nonlinear equations expressed as constraints between numeric variables. The function `nonlinear` finds a solution to the following system of nonlinear equations.

$$\begin{aligned} 4x^2y + 7yz^2 + 6x^2z^2 &= 1356.14 \\ 3xy + 2y^2 + 5xyz &= -141.375 \\ (x+y)(y+z) &= -7.7625 \end{aligned}$$

```
(defun nonlinear ()
 (one-value
  (let ((x (a-real-betweenv -1e40 1e40))
        (y (a-real-betweenv -1e40 1e40))
        (z (a-real-betweenv -1e40 1e40)))
   (assert!
    (andv (=v (+v (*v 4 x x y) (*v 7 y z z) (*v 6 x x z z)) 1356.14)
          (=v (+v (*v 3 x y) (*v 2 y y) (*v 5 x y z)) -141.375)
          (=v (*v (+v x y) (+v y z)) -7.7625)))
   (solution (list x y z)
             (reorder #'range-size
                      #'(lambda (x) (< x 1e-6))
                      #'>
                      #'divide-and-conquer-force)))))
```

Figure 2: A SCREAMER program which uses numeric bounds propagation for solving a system of nonlinear equations.

The expression (`a-real-betweenv -1e40 1e40`) creates a variable constrained to be a real number between the given upper and lower bounds. After the constraints have been asserted between the variables, divide and conquer search—interleaved with bounds propagation—is used to find a solution to the equations. One such solution is $x \approx -7.311$, $y \approx 6.113$, $z \approx 0.367$. Note that unlike the simplex method used in CLP($\Re$)—which is limited to solving linear systems of equations—the combination of divide and conquer search interleaved with bounds propagation allows SCREAMER to solve complex nonlinear systems of equations. These techniques also enable SCREAMER to solve numeric constraint systems with both inequalities and equational constraints. Furthermore, since all of the constraint satisfaction techniques are integrated, SCREAMER can solve disjunctive systems of equations as well as systems which mix together numeric, Boolean, and other forms of constraints.

The third example solves crossword puzzles. (The code for this example was eliminated due to space restrictions but will be included in the full paper.) Here, a crossword puzzle is represented as a list of placements of the form (i j `across` l) or (i j `down` l) denoting an across or down entry for a word of length l starting at row i and column j. The dictionary is a list of words. The objective is to fill each placement with a word from the dictionary such that the crossword puzzle is consistent. (Different placements may utilize the same word.) The program operates by creating a variable for each placement. The domain of each variable is restricted to those words from the dictionary of the appropriate length. Further pairwise constraints are added between intersecting placements which restrict the values of those variables to share the same letter at the point of intersection. This program can solve crossword puzzles from typical games magazines—where the dictionary is taken to be the list of words appearing in the solution—in a few seconds of CPU time.

We wish to point out the intentional similarity in the names of the SCREAMER primitives `a-member-of` and `a-member-ofv`. Both describe a choice between a set of possible alternatives. The former enumerates that set nondeterministically by backtracking. The latter instead, creates a variable whose value is constrained to be a member of the given set. The former lends itself to a generate-and-test style of programming:

```
(let ((x1 (a-member-of s1)) ... (xn (a-member-of sn)))
 (unless Φ[x1...xn] (fail))
 (list x1...xn))
```

The latter lends itself to constraint-based programming:

```
(let ((x1 (a-member-ofv s1)) ... (xn (a-member-ofv sn)))
 (assert! (funcallv #'(lambda (x1...xn) Φ[x1...xn]) x1...xn))
 (solution (list x1...xn) (static-ordering #'linear-force)))
```

Though these two program fragments are structurally very similar, and specify the same problem, they entail drastically different search strategies. The latter constitutes a lifted variant of the former. A future paper will we discuss the possibilities of performing such lifting transformations automatically. Such lifting is not limited to the `a-member-of` primitive. SCREAMER includes the following syntactic sugar for `(an-integer-betweenv` l h`)`:[4]

```
(let ((v (make-variable)))
 (assert! (andv (integerpv v) (<=v v h) (>=v v l)))
 v)
```

The function `an-integer-betweenv` is a lifted analog to the SCREAMER primitive `an-integer-between`. All SCREAMER generators have lifted analogs.

6 Related Work

Most of the individual techniques described in this paper are not new. What is novel is their particular combination. Programming languages which allow stating numeric constraints date back to SKETCHPAD (Sutherland 1963). Local propagation for solving systems of constraints was used by Borning (1979) in THINGLAB. Steele (1980) and Sussman and Steele (1980) construct constraint primitives very similar to ours and implement local propagation by procedural attachment. These techniques were expanded on by the MAGRITTE

[4] We adopt the (unenforced) convention that functions beginning with the prefix `a-` or `an-`, and ending with `v`, denote lifted generators, functions which return a variable ranging of the stated domain instead of nondeterministically returning a value in that domain.

system (Gosling 1983). The above systems differ from SCREAMER in two ways. First, they handled only numeric constraints, lacking the GFC capacity of SCREAMER embodied in `memberv` and `funcallv`, as well as unification embodied in `equalv`. More importantly, the constraint solving techniques incorporated in all of these systems were incomplete, particularly those based on local propagation. None of these systems could resort to backtracking search—as SCREAMER can—to provide a slow but complete fallback to faster but incomplete local propagation techniques.

More recently, numerous systems such as CLP($\Re$) (Jaffar and Lassez 1987) and CHIP (Van Hentenryck 1989) have been constructed in the logic programming framework, which add some form of constraint satisfaction—sometimes based on local propagation—to the backtracking search mechanism already present in logic programming languages. SCREAMER differs from such systems in a number of ways, some minor and some major. First, SCREAMER uses only fast local propagation techniques as part of its constraint mechanism. The numeric constraint mechanism of CLP($\Re$) instead uses more costly techniques based on the simplex method for linear programming. These techniques break down for nonlinear constraints. SCREAMER, on the other hand, can solve nonlinear constraints using `divide-and-conquer-force` combined with local propagation. It turns out that such techniques are a generalization of ones previously proposed as good numerical analysis techniques. The second difference lies in using COMMON LISP instead of PROLOG as a substrate for constructing constraint-based programming languages. As mentioned above, COMMON LISP alone is inadequate as a basis for constructing constraint languages due to its lack of nondeterminism as a general fallback technique. This has been remedied in SCREAMER via the implementation of nondeterministic COMMON LISP. Given the substrate of nondeterministic COMMON LISP—especially its capacity for local side effects—the SCREAMER constraint package can be written totally in COMMON LISP, with no significant loss of efficiency. This is in contrast to CLP($\Re$) and CHIP, which cannot be efficiently implemented as extensions to PROLOG, but rather require completely new implementations in the host language. This gives SCREAMER three clear advantages over CLP($\Re$) and CHIP, over and above the stylistic benefits of functional versus logic programming. First, SCREAMER is portable to any COMMON LISP implementation. Second, SCREAMER can be easily modified and extended, to experiment with alternative constraint types and constraint satisfaction methods. Finally, SCREAMER can coexist and inter-operate with other current or future extensions to COMMON LISP such as CLOS and CLIM. This allows nondeterminism and constraint-based programming to be just one more tool available to the programmer in her bag of tricks, instead of a pervasive paradigm which can strangle and stifle programming creativity.

The current version of SCREAMER, including the full constraint package, is available by anonymous FTP from the file `ftp.ai.mit.edu:/com/ftp/pub/screamer.tar.Z`. We encourage you to obtain a copy of SCREAMER and give us feedback on your experiences using it.

References

[1] Alan Hamilton Borning. THINGLAB—*A Constraint-Oriented Simulation Laboratory.* PhD thesis, Stanford University, July 1979. Also available as Stanford Computer Science Department report STAN-CS-79-746 and as XEROX Palo Alto Research Center report SSL-79-3.

[2] James Gosling. *Algebraic Constraints.* PhD thesis, Carnegie-Mellon University, 1983.

[3] E. R. Hansen. On the solution of linear algebraic equations using interval arithmetic. *Mathematical Computation*, 22:153–165, 1968.

[4] E. R. Hansen and G. W. Walster. Nonlinear equations and optimization. *Control and Games of Comput. Math. Appl.*, 1991. To appear in the second special issue on Global Optimization.

[5] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Unknown*, pages 111–119, 1987.

[6] Alan K. Mackworth. Constraint satisfaction. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 285–293. John Wiley & Sons, Inc., New York, 1992.

[7] David Allen McAllester. Truth maintenance. In *Proceedings of the Eighth National Conference on Artifical Intelligence*, pages 1109–1116. Morgan Kaufmann Publishers, Inc., 1990.

[8] Jeffrey Mark Siskind and David Allen McAllester. SCREAMER: a portable efficient implementation of nondeterministic COMMON LISP. Submitted to AAAI-93.

[9] Ivan E. Southerland. SKETCHPAD*: A Man-Machine Graphical Communication System.* PhD thesis, Massachusetts Institute of Technology, January 1963.

[10] Guy Lewis Steele Jr. *The Definition and Implementation of a Computer Programming Language Based on Constraints.* PhD thesis, Massachusetts Institute of Technology, August 1980. Also avilable as M. I. T. VLSI Memo 80-32 and as M. I. T. Artificial Inteligence Laboratory Technical Report 595.

[11] Gerald Jay Sussman and Guy Lewis Steele Jr. CONSTRAINTS—a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14(1):1–39, 1980. Also available as M. I. T. Artificial Intelligence Laboratory Memo 502A.

[12] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* M. I. T. Press, Cambridge, MA, 1989.