Local Propagation as a Constraint Satisfaction Technique

John Mulac
Doug Baldwin

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 265
January 1989

Abstract

Motivated by the problem of programming multiprocessors, we study constraint satisfaction as a paradigm on which practical, executable, programming languages can be based. Others have described constraint-based languages, but little analysis of constraint satisfaction heuristics seems to have been done. The contributions of this paper are twofold. First, we examine the complexity of the constraint satisfaction problem. Second, and most important, a formalization of a local propagation heuristic that simplifies one used earlier by Steele [12] is described.

This work was supported by the National Science Foundation under grants number DMC-8613489 and DCR-8320136, and by the U.S. Army Engineering Topographic Laboratories under grant number DACA76-85-C-0001.
Local Propagation as a Constraint Satisfaction Technique

John Mulac and Doug Baldwin

Department of Computer Science
734 Computer Studies Bldg.
Univ. of Rochester, Rochester, NY 14627

Motivated by the problem of programming multiprocessors we study constraint satisfaction as a paradigm on which practical, executable, programming languages can be based. Others have described constraint-based languages, but little analysis of constraint satisfaction heuristics seems to have been done. The contributions of this paper are twofold. First, we examine the complexity of the constraint satisfaction problem. Second, and most important, a formalization of a local propagation heuristic that simplifies one used earlier by Steele is described.
1 Introduction

Programming multiprocessors in ways that effectively use their parallelism is widely felt to be a difficult job. One reason is that none of the programming languages presently proposed for this work adapts well to general-purpose parallel programming [2]. Of all the existing approaches, logic programming seems to be the most promising. While pure logic programming has a number of features that seem well-suited to parallelization, practical realizations are forced to introduce a number of extra-logical features that interfere with parallelization. The reason these features are introduced is that the first-order predicate calculus on which the languages are based is not powerful enough to describe common operations such as arithmetic, input and output, et cetera in computationally practical ways. One promising approach to solving this problem is to generalize logic languages to constraint languages. As with logic languages, a program in a constraint language is a system of relations that is "executed" by mechanically proving that certain variables (the outputs) must have certain values as a consequence of the constraints imposed by the system. Unlike logic languages, relations in a constraint language can be expressed in formalisms other than predicate calculus. An early constraint language is described in [12]. Recently there has been a flurry of interest in constraint languages, much of it inspired by attempts to extend the present state of the art in logic programming [3; 5; 7; 9]. Other recent work has been motivated by work on term rewriting as a mechanism for solving systems of equations [6; 10].

One of the major reasons why logic languages are based on a formalism with limited practical expressive power is that efficient methods exist for proving theorems within this formalism. By accepting a richer formalism as primitive, general constraint languages usually require ad hoc heuristic satisfaction methods in order to make implementations practical. Most published descriptions of constraint languages contain few formal analyses of the constraint satisfaction heuristics used in their implementations. This paper first examines the complexity of constraint satisfaction and then describes a schema for building heuristics to prove satisfiability or unsatisfiability of systems of constraints based on very local proofs of individual constraints. The basic schema was created for the CONSUL language [1], but should be readily adaptable to other constraint languages.

2 Constraint Satisfaction — General Complexity

Intuitively, a constraint defines a property that holds for some variables. For example, \((X > 1)\) is a constraint that states that variable \(X\) has a value greater than one. As this example shows, a variable may have more than one value that satisfies a constraint, thus a variable represents a value from a set of possible values. Constraints joined by logical connectives form a system of constraints.

Heuristic approaches to constraint satisfaction are necessary because determining the satisfiability of a system of constraints is an unsolvable problem. To see this result, we make the following definitions.

Definition 1 A variable is the name for a set of values.
Definition 2 A constraint is a relation between values.

Definition 3 A constraint system is a set of constraints.

Definition 4 The constraint satisfaction problem is the following decision problem: Given a constraint system, do there exist values for all variables in this system such that all constraints in the system simultaneously hold true?

Matijasevič [11] proved the unsolvability of Hilbert’s Tenth Problem. In this proof, he used the notions of a diophantine equation and a recursively enumerable predicate. A diophantine equation is an equation of the form \( P(x_1, \ldots, x_n) = 0 \), where \( P \) is a polynomial with integer coefficients and the \( x_i \) range over the integers. A recursively enumerable (r.e.) predicate for a set \( S \) is a function that returns 0 if its argument is not an element of \( S \), otherwise it returns some non-zero value. Matijasevič showed that every r.e. predicate can be expressed as a diophantine equation. Since there exist undecidable r.e. predicates, e.g., the Halting Problem, Matijasevič’s result implies there exist undecidable diophantine equations. Therefore, for constraint languages over the integers that allow addition and multiplication, no algorithm exists that can determine the satisfiability of all systems of constraints. In particular, CONSUL, one such constraint language, exhibits this property.

As an aside, this leads to an interesting question: Is the Church-Turing thesis wrong? Given the semantics of constraint languages like CONSUL, “correct” programs can be written whose satisfiability is not determinable by a Turing machine. For example, one could easily construct a constraint program that solves the Halting Problem. This seemingly contradicts the Church-Turing thesis. However, we do not believe the Church-Turing thesis is invalid since constraint programs are not algorithms describing how to solve problems but rather descriptions of what the solutions “look like”. Furthermore, any current implementation of a constraint language could not “execute” these programs to completion on all inputs.

The above shows that for any constraint language that allows addition and multiplication over the integers there does not exist an algorithm to solve all possible constraint systems. This leads to the question: If the programming language is restricted, can all constraint systems be solved algorithmically? If one restricted oneself to the sub-class of constraint languages called the logic programming languages, the answer would still be “no”. For example, PROLOG is one such language. The reason that PROLOG does not have a satisfaction algorithm is that PROLOG functors can be used to express Peano arithmetic. Therefore, diophantine equations can be written as valid PROLOG programs, implying no general algorithm to solve all systems of constraints can exist. Rather, PROLOG implementations use resolution theorem-proving in a Horn clause environment and resolution is only refutation complete. This example also shows one of the difficulties in categorizing constraint languages: “hard” features such as addition and multiplication can be derived from much “simpler” primitives such as functors.

\(^1\)Hilbert’s Tenth Problem is: Given a diophantine equation with any number of unknown variables and with integral numerical coefficients, does there exist an algorithm to determine if the equation is solvable in the integers?
If a language has only boolean variables, connectives and quantifiers, then the satisfaction problem is PSPACE-complete because any program in this language is an instance of the Quantified Boolean Formulas (QBF) problem, a known PSPACE-complete problem [13]. Restricting a language to only boolean variables and connectives means satisfaction would be NP-complete, since any program is an instance of the Satisfiability (SAT) problem, the first problem proven to be NP-complete [4]. Thus, significant restrictions cut expressive power but still require a heuristic to efficiently solve programs.

In this section we have described the complexity of the constraint satisfaction problem and outlined a hierarchy of constraint systems. Similar results have been observed by others. The importance of Matijasević's result was recognized by Goguen [5] and a less general hierarchy of constraint systems was noted by Van Wyk [14].

3 The Satisfaction Heuristic

The previous section shows that there is no practical algorithm to solve all constraint systems written in any useful constraint language. However, heuristics exist to solve some systems of constraints and we will formally examine one such heuristic in this section. Our heuristic is inspired by Steele's local propagation technique [12]. We begin with an informal description of the heuristic.

Consider a constraint program to be a number of atomic primitive constraints joined into a larger system by logical connectives and quantifiers. Local propagation proves or disproves satisfiability of a system of constraints by proving each of its primitive constraints. For each type of primitive constraint there is a procedure to prove or disprove instances of the constraint. At the time such a procedure is invoked on a particular constraint instance, some of the arguments to the constraint are already bound (by bound, we mean the arguments have known values), either as a result of earlier satisfiability proofs or because they were constants to begin with. Other arguments are unbound. The satisfaction procedure simply tests the pattern of bound and unbound arguments, and dispatches to a sub-procedure (henceforth called a method) that either computes satisfying values for one or more unbound arguments, or tests bound arguments to see if the constraint is satisfied. The exact patterns tested and methods invoked reflect the algebraic properties of the constraint being proved. For example, Figure 1 shows pseudo-code for a satisfaction procedure for constraints of the form "\(X = Y + Z\)". The results of these local proofs are communicated between satisfiers in order to arrive at a consistent set of variable-to-value bindings that satisfies the system as a whole. One important distinction between our approach to local propagation and Steele's is that we allow variables to be bound to sets of values rather than just to individuals. This feature allows the final value (or values) of a variable to be the joint result of several proofs without explicit backtracking or retraction mechanisms. The exact pattern of communication between primitive satisfiers is determined by the connectives or quantifiers joining the corresponding constraints. Because individual satisfiers never contradict bindings established by other satisfiers, the final set of bindings (if any) produced by such a network does in fact satisfy all of the constraints.

\(2\)This is an exact description of CONSUL programs, and a general description that all other constraint
To show satisfiability/unsatisfiability of \( X = Y + Z \):

- if \( X \) is unbound, \( Y \) and \( Z \) are bound
  - Satisfiable, assign \( Y + Z \) to \( X \).
- else if \( Y \) is unbound, \( X \) and \( Z \) are bound
  - Satisfiable, assign \( X - Z \) to \( Y \).
- else if \( Z \) is unbound, \( X \) and \( Y \) are bound
  - Satisfiable, assign \( X - Y \) to \( Z \).
- else if \( X, Y, \) and \( Z \) are all bound
  - if \( X = Y + Z \)
    - Satisfiable, no new bindings needed.
  - else
    - Unsatisfiable.
- else
  - Not enough information yet for proof.

Figure 1: Outline of Satisfier for Primitive “Sum” Constraints

Now, we develop the satisfaction heuristic that attempts to solve systems of constraints. Formally, we develop the heuristic for conjunctions, then explain how to extend it to include disjunctions, negations and quantifiers. This heuristic is based on the following definitions.

**Definition 5** The variable set of a constraint is the set of variables that appear in it. The variable set of a set of constraints is the union of the variable sets of the individual constraints.

**Definition 6** Variables in a set of constraints represent values from a set called the value space of the constraints.

**Definition 7** An environment is a total mapping from the variable set of a set of constraints to the power set of its value space.

Intuitively, an environment of a system of constraints associates a set of values with each variable. The overall goal of the satisfaction heuristic will be to find an environment for a system such that variables are associated only with values that satisfy all constraints. Note that some systems may have multiple satisfying environments. We formalize the idea of constraints being “satisfied” as follows:

**Definition 8** A constraint \( (Cx_1x_2 \ldots x_n) \) is satisfied by environment \( E \) if and only if for all variables \( x_i \) in the variable set of \( C \) and all substitutions of values \( v_i \in E(x_i) \) for \( x_i \), the tuples \( [v_1, \ldots, v_n] \) are elements of the set denoted by \( C \). A system of constraints is satisfied...
Given constraint \( c \) and an environment \( E \)
select a \( p^i_\alpha \) such that \( p^i_\alpha(E) = \text{TRUE} \)
\[ E' = F^i_\alpha(E) \]

Figure 2: General action of a primitive satisfier for a constraint \( c \).

by \( E \) if and only if all constraints in the system are satisfied by all substitutions of values from \( E \) for variables in the system. A constraint or system of constraints is \( \text{satisfiable} \) if it is satisfied by some environment.

Note that all constraints must be satisfied; therefore, the constraints within a system are implicitly joined by a conjunction. The existence of an environment in which no variable is mapped to the empty set constitutes evidence that a system is satisfiable.

With respect to Definition 8, note that \( \text{all substitutions} \) must be elements of the set \( C \). For example, the system \( ((X > 0) \land (Y = X + 2)) \) is \( \text{not satisfied} \) by the environment \( (X = \{v : v > 0\}, Y = \{v : v > 2\}) \) because the substitutions of 1 for \( X \) and 4 for \( Y \) yields \( 5 = 1 + 2 \) which is not true. Rather, this example would have multiple satisfying environments: \( ((X = \{1\}, Y = \{3\}), (X = \{2\}, Y = \{4\}), ...) \).

Local propagation attempts to transform an initial environment that does not satisfy a system of constraints into one that does. The transformation is achieved by applying a primitive satisfier for some constraint in the system to the initial environment, producing an intermediate environment that reflects the new bindings created by the primitive satisfier. A primitive satisfier for another constraint is applied to this intermediate environment, and so forth, until the final environment is produced. The primitive satisfiers are specific to the particular algebra underlying the system of constraints. We thus adopt a "black box" formalization of primitive satisfiers, allowing us to characterize them well enough to develop a formal description of their use without requiring so much detail on their internal mechanisms as to compromise the generality of our heuristic.

Specifically, we view local satisfiers for constraints as finite sets of paired predicates and functions \( \{(p_1, F_1), (p_2, F_2), ..., (p_n, F_n)\} \). The predicates map environments to truth values; the functions map environments to environments. The predicates represent the tests for patterns of bound and unbound variables in the constraints' variable sets; the functions represent the associated methods. Figure 2 describes the overall action of a primitive satisfier, producing environment \( E' \) from environment \( E \) given a constraint \( c \); the \( p^i_\alpha \) notation represents the \( i^{th} \) predicate \( p \) of the local satisfaction method for constraint \( c \).

The \( F^i_\alpha \) within a primitive satisfier transform environments by changing the sets to which environments map certain variables. The new sets will be functions of the values to which certain other variables were mapped in the original environment. More precisely, if \( F^i_\alpha \) is a method that produces \( E' \) from \( E \) for constraint \( c \), then \( E' = E[x_j/f_j(E(x^i_1), E(x^i_2), ..., E(x^i_n))] \) for all \( x_j \) in some subset of \( c \)'s variable set. For each \( x_j \) there is a distinct \( f_j \) (a sub-function of \( F^i_\alpha \)) that generates the new mapping for \( x_j \). Each \( f_j \) may be a function of (the mappings of) a distinct set of variables (the \( x^i_\alpha \)).
local_propagate \((C,E)\)
while \(C \neq \emptyset\)
  \(E' \leftarrow E\)
  \(C' \leftarrow \emptyset\)
  for all \(c \in C\) such that \(\exists p_c : p_c(E)\)
    \(E' \leftarrow E' \text{ merge } F_c^c(E)\)
    \(C' \leftarrow C' \cup \{c\}\)
  \(C \leftarrow C - C'\)
  \(E \leftarrow E'\)
return \(E\)

Figure 3: The local propagation heuristic.

Figure 3 shows the local propagation heuristic. The heuristic works as follows: For all constraints that have a predicate \(p_c\) that is true for the current environment, call the corresponding environment function \(F_c^c\). Apply a merge operation (defined below) to the initial environment for this iteration of the while loop and the environments produced by the \(F_c^c\) in order to resolve any inconsistencies between a variable's values in these environments. Then, use this new environment in the next iteration to "solve" the remaining constraints. If no predicate for any remaining constraint holds for the current environment, satisfiability of the system is not provable using local propagation. This heuristic relies on certain assumptions about satisfaction methods. These assumptions are described using the following definitions.

Definition 9 Environment \(E_1\) is a restriction of \(E_2\) if and only if \(E_1\) and \(E_2\) have the same domain, and for all variables \(X\) in this domain, \(E_1(X) \subseteq E_2(X)\).

Definition 10 \(E\) is a maximal environment with respect to predicate \(P\), if no variable \(x\) mapped by \(E\) can be mapped to a set bigger than \(E(x)\) without the resulting environment either violating \(P\) or mapping some other variable, \(y\), to a set different from \(E(y)\).

Predicate \(P\) in the above definition will typically be something like \(E\) being a satisfying environment, \(E\) being a restriction of another environment, et cetera.

The assumptions about satisfaction methods are as follows. Here, \(E'\) represents the output environment produced by a satisfaction method; \(E\) represents the corresponding input environment.

1. The constraints and environments involved in proofs are such that for each constraint, there is a unique maximal restriction of \(E\) that satisfies the constraint. We will explain how to relax this assumption later.

2. \(E'\) is the maximal satisfying restriction of \(E\).
3. The implementations of all $p_i$ in a satisfaction method terminate. Furthermore, for all pairs $(p_i, F_i)$ in a satisfaction method, whenever $p_i$ is true for an environment, the implementation of $F_i$ terminates.

The **merge** operation applied to a pair of environments produces a new environment whose domain is the intersection of domains of the original environments and in which each variable in this domain is mapped to the intersection of the sets to which that variable was mapped by the original environments. Hence, given environments $E_1$ and $E_2$, if $E'$ equals $E_1$ merge $E_2$, then for all variables $X$ common to $E_1$ and $E_2$, if $E_1(X) = S_1$ and $E_2(X) = S_2$, then $E'(X) = S_1 \cap S_2$.

Our first theorem establishes a necessary condition for termination of $\text{local-propagate}$; the heuristic can easily test this condition as it executes.

**Theorem 1** If $\text{local-propagate}$ terminates, then at the beginning of each iteration of the while loop at least one of the constraints remaining to be solved has a local satisfier's predicate that evaluates to true.

**Proof:** Follows immediately from the definition of $\text{local-propagate}$ and the assumption that local satisfaction methods always terminate. \(\square\)

We now introduce lemmas that will be used in proving our next theorem.

**Lemma 1** Let $E$ be an environment. Let $E_1$ be the unique maximal restriction of $E$ satisfying constraint system $C_1$. Let $E_2$ be the unique maximal restriction of $E$ satisfying constraint system $C_2$. Then, if $E'$ equals $E_1$ merge $E_2$ and $C'$ equals $C_1 \cup C_2$, then $E'$ is the unique maximal restriction of $E$ satisfying constraint system $C'$.

**Proof:** First, we show that $E'$ is a restriction of $E$. Since $E_1$ and $E_2$ are restrictions of $E$, then $E$, $E_1$ and $E_2$ have the same domain. Therefore, by the definition of merge, $E'$ has the same domain as $E$; furthermore, for all variables in this domain, $E'(X) \subseteq E_1(X)$. By the definition of restriction, $E_1(X) \subseteq E(X)$; hence, $E'(X) \subseteq E(X)$ implying $E'$ is a restriction of $E$. Now, we show that $E'$ is a restriction of $E$ satisfying constraint system $C'$. By $E''$'s definition, any value in $E'(X)$ is in both $E_1(X)$ and $E_2(X)$. Values in $E_1(X)$ satisfy constraint system $C_1$ by assumption and similarly values in $E_2(X)$ satisfy constraint system $C_2$; thus, all values in $E'(X)$ satisfy both $C_1$ and $C_2$. Therefore, all values in $E'(X)$ satisfy $C_1 \cup C_2$, which equals $C'$, implying $E'$ is a restriction of $E$ satisfying $C'$. Next, we show that $E'$ is a maximal restriction of $E$ satisfying $C'$. Suppose not, then for some variable $x$ there exists a value of $x$, call it $v_x$, such that $v_x \notin E'(x)$ but $v_x$ satisfies $C'$ without remapping any other variable in $E'$. Because $v_x$ satisfies $C'$, $v_x$ satisfies $C_1$. This implies $v_x$ is an element of some maximal environment that satisfies $C_1$. However, $E_1$ is the only such environment, implying $v_x \in E_1(x)$; similarly for $E_2$ satisfying $C_2$, we obtain $v_x \in E_2(x)$. This implies $v_x \in E'(x)$, a contradiction; therefore, $E'$ must be a maximal restriction satisfying $C'$. Finally, we show that $E'$ is the unique maximal restriction of $E$ satisfying $C'$. Suppose there exists another environment $E''$ which is a maximal restriction of $E$ satisfying $C'$. This implies that $E''$ is a restriction of $E$ satisfying $C_1$ and satisfying
Lemma 2 Let $E$ be an environment. Let $E_1$ be the unique maximal restriction of $E$ satisfying constraint system $C_1$ and let $E_2$ be the unique maximal restriction of $E_1$ satisfying constraint system $C_2$. Then, $E_2$ is the unique maximal restriction of $E$ satisfying constraint system $C'$ equal to $C_1 \cup C_2$.

Proof: First, we show that $E_2$ is a restriction of $E$ satisfying $C'$. By the definition of restriction, $E_2$ and $E$ have the same domain. Furthermore, for all variables $X$, $E_2(X) \subseteq E_1(X)$ and $E_1(X) \subseteq E(X)$ implying $E_2(X) \subseteq E(X)$. Also, since $E_2(X) \subseteq E_1(X)$, then all values in $E_2(X)$ satisfy $C_1$ and, by definition, satisfy $C_2$; therefore, $E_2$ is a restriction of $E$ satisfying $C'$. Next, we show that $E_2$ is a maximal restriction of $E$ satisfying $C'$. Suppose not, then for some variable $x$ there exists a value of $x$, call it $v_x$, such that $v_x \in E_2(x)$ but $v_x$ satisfies $C'$ without remapping any other variable in $E_2$. Since $v_x$ satisfies $C'$, $v_x$ satisfies $C_1$. This implies $v_x$ is an element of some maximal environment that satisfies $C_1$. However, $E_1$ is the only such environment, implying $v_x \in E_1(x)$. Since $v_x$ satisfies $C_2$ and $v_x \in E_1(x)$, then, by maximality of $E_2$ with respect to $C_2$, $v_x$ must be in $E_2(x)$, a contradiction; therefore, $E_2$ must be a maximal restriction satisfying $C'$. Finally, we show that $E_2$ is the unique maximal restriction of $E$ satisfying $C'$. Suppose there exists another environment $E'$ which is a maximal restriction of $E$ satisfying $C'$. This implies that $E'$ is a restriction of $E$ satisfying $C_1$. Since $E_1$ is the unique maximal restriction of $E$ satisfying $C_1$, then for all variables $X$, $E'(X) \subseteq E_1(X)$. Since $E'(X) \subseteq E_1(X)$ and all values in $E'(X)$ satisfy $C_2$, then $E'(X) \subseteq E_2(X)$ for all variables $X$ because $E_2$ is the unique maximal restriction of $E_1$ satisfying $C_2$. However, if $E'(X) \subseteq E_2(X)$, then $E'$ is not maximal, a contradiction. Therefore, $E_2$ is the unique maximal restriction of $E$ satisfying $C_1 \cup C_2$. □

Lemma 3 Given a constraint system $C$ and an initial environment $E$, let $C_n$ be the conjunction of $C$'s constraints solved in the first $n$ iterations of the while loop in local_propagate and let $E_n$ be the environment produced after the $n$th iteration. Then, $E_n$ is the unique maximal restriction of $E$ satisfying constraint system $C_n$.

Proof: By induction on $n$. Base Case: $n = 0$. This means $C_n = \emptyset$ and $E_n = E$, thus trivially, $E_n$ is the unique maximal restriction of $E$ satisfying $C_n$. Induction step: Assume the lemma is true for $n \leq k$, we will prove the lemma for $n = k + 1$. First, run local_propagate for $n - 1$ iterations. By the induction hypothesis, the resulting environment $E_{n-1}$ is the unique maximal restriction of $E$ satisfying $C_{n-1}$ (all constraints satisfied in the first $n-1$ iterations). Now, let us define $C_*$ to be the set of constraints in $C - C_{n-1}$ that have at least one of the their local satisfier's predicates true for environment $E_{n-1}$. We have two possible cases. First, $C_*$ = $\emptyset$ implying $C_n = C_{n-1}$ and $E_n = E_{n-1}$; hence, by the induction hypothesis, $E_n$ is the unique maximal restriction of $E$ satisfying $C_n$. Second, $C_* \neq \emptyset$. Therefore, for all constraints $c_j$ in $C_*$, the appropriate mapping functions are called to obtain new
environments $E_{c_j}$. Each $E_{c_j}$ is the unique maximal restriction of $E$ satisfying $C_{n-1} \cup \{c_j\}$ (by Assumptions 1 and 2 about the satisfaction methods and by Lemma 2). Thus, after the final incremental merge of the $E_{c_j}$ occurs, the resulting environment $E_n$ is the unique maximal restriction of $E$ satisfying $C_n$ equal to $C_{n-1} \cup C_s$ (follows from Lemma 1).

We may now state and prove the following theorem about the local propagation heuristic.

**Theorem 2** Given a constraint system $C$ and an initial environment $E$, if local-propagate terminates, then the environment that it returns is the unique maximal restriction of $E$ that satisfies $C$.

**Proof:** Since the algorithm terminates, there must be a finite number of iterations of the while loop after which all constraints have been solved. Thus, by Lemma 3, the environment produced after these iterations is the unique maximal restriction of $E$ satisfying $C$. □

Informally, Theorem 2 says that if local propagation is applied to a solvable system of constraints, then it will either find all solutions to that system or will fail to terminate. The maximal restrictions that "satisfy" an unsolvable system of constraints are those that map one or more variables to the empty set. Thus local-propagate also proves unsolvability of unsolvable systems (or doesn't terminate).

Theorem 1 is important for its uses in demonstrating when local propagation does not work. For example, one weakness of local propagation is that it cannot handle a simultaneous system of equations. By the algebraic properties that make the equations "simultaneous", none of the predicates of the local satisfiers can evaluate to true when applied to the initial environment (otherwise one of the local satisfiers could solve a constraint without information from other constraints). The inability of local propagation to solve such systems then follows immediately from the theorem. Consider, for example, the following set of constraints: $(X + Y = 2) \land (X - Y = 0)$ for $X$ and $Y$ elements of the integers. Obviously, the solution of each constraint depends on that of the other, but local propagation lacks a mechanism to describe this dependency.

The ability to represent the values of variables as sets provides a "hook" via which routines for solving simultaneous systems of equations can be integrated with local propagation. The central idea is to allow the option of representing sets of values by the constraints that the values must satisfy. A satisfier that cannot determine specific values for variables can then attach the relevant constraint to the variables in question, postponing exact solution until later. Variables may continue to collect attached constraints; eventually these constraints are passed to a routine for solving simultaneous systems that computes (if possible) the exact values of the variables. The simultaneous system solver can be based on any of a number of algorithms, for example gaussian elimination (if the systems are linear) or relaxation. The decision to invoke the simultaneous system solver can be made in any of several ways: try it whenever a new constraint is attached to a variable, call it after everything has been satisfied by local propagation that can be, et cetera. This technique is similar to Jaffar and Michaylov's tiered approach to constraint satisfaction in a Constraint Logic Programming System [8]. Local propagation with sets of values is thus a good primary satisfaction method, since it is very fast, seems to handle most real programs, and has a promising hook for accessing secondary satisfaction methods when they are needed.
Previously, it was noted that the local propagation heuristic was developed for conjunctions. We will discuss how the heuristic is extended for disjunctions, negations and quantifiers. For disjunctions, the idea is simply to copy the environment for each branch of the disjunction and allow local propagation to work on each branch separately. Any branch that succeeds will have a corresponding environment that is one of possibly many independent solutions to the disjunction. Figure 4 illustrates an example disjunction. To handle negations, the heuristic attempts to prove the body of the negation; if the body is proved then the negation is disproved, otherwise if the body is disproved the negation is proved. To handle the universal quantifier over finite sets, an environment is produced with the quantified variable mapped to a single value and the body of the universally quantified expression is satisfied in this environment. Repeating for other values in the range of the quantified variable gives a set of environments that can be merged. The resulting environment satisfies the universally quantified expression. Figure 5 portrays an example of a universally quantified expression. To handle the existential quantifier, the idea is the same except that the resulting environments need not be merged, each environment that does not map any variable to the empty set is passed on as one of the possibly many solutions to the existentially quantified expression.

We now relax two of our other assumptions. First, when we introduced the satisfaction

---

\(^3\)The observant reader will note that this results in the quantified variable being mapped to the empty set, implying unsolvability of the system. Therefore, a modified version of merge is used for universal quantification that maps the quantified variable to the union of the sets to which that variable was mapped in each environment, but for all other variables acts the same as the original merge.
for all $x$ in $1 .. n$, prove $C$

**Figure 5:** The method to handle universal quantification over constraints.

heuristic, we assumed that every constraint had a unique maximal satisfying environment. However, constraints can have multiple maximal satisfying environments. Figure 6 depicts how a modified version of the heuristic would handle the situation when one constraint has two maximal satisfying environments. The idea is that each maximal environment produced by `local.solve` is merged with the environments produced from solving the other constraints in that iteration of the `while` loop (the LS nodes in Figure 6 represent local solvers and the e arcs represent the environments produced by these solvers). Then, the resulting environments ($E1$ and $E2$ in the figure) are treated as independent environments; each of these environments along with its own copy of the remaining constraints ($C - C'$) is passed to the next iteration of the `while` loop in `local.propagate` (the LP nodes in the figure). Note that this scheme is similar to the method we outlined for handling disjunctions. Thus, to each "forked" LP, the constraints to solve and the input environment resemble any other system `local.propagate` would attempt to solve. Nothing is different; none of the lemmas or theorems are affected. The only difference is that all the forked LP’s should be considered as connected by an `or` in the sense that if any of the forked LP’s succeed, then the original system is satisfiable.

We also relax the assumption that for each iteration of the `while` loop, `local.propagate` merges all environments resulting from satisfaction of a constraint to produce the initial environment for the next iteration. This restriction can be relaxed so that the local solvers merge only those environments that they need to solve their particular constraint. The only additional requirement is that a final merge of environments must occur in order to find any inconsistencies. Figure 7 shows a simple example where `output` is the set of
variables that are bound by the solution of a constraint and input is the set of variables that must be bound to solve a constraint. In this example, to solve constraint $c_5$, only variables $W$ and $X$ must be bound; these variables are bound by constraints $c_1$ and $c_2$, respectively. Thus, only the environments produced by $c_1$ and $c_2$ need to be merged before $c_5$ can be solved. Similarly for solving $c_6$, only the environments resulting from $c_3$ and $c_4$ need to be merged. The final (rightmost) merge resolves any inconsistencies between the environments resulting from the solutions of $c_5$ and $c_6$.

Note that parallelism in local propagation arises from the fact that, in general, proofs of any two constraints in one iteration of the while loop can be attempted concurrently. Distributing the merges reduces bottlenecks due to solvers synchronizing and merging environments after each iteration, and may allow some concurrency between iterations.

### 4 Summary and Conclusions

This paper has studied constraint satisfaction as a paradigm on which practical, executable, programming languages can be based. Others have described constraint-based languages, but little analysis of constraint satisfaction heuristics has been done. This paper offers one formal analysis of a local propagation heuristic that simplifies one used earlier by Steele. In Theorem 1 we have shown a necessary and useful condition for local propagation to terminate. Furthermore, Theorem 2 proves that when local propagation does terminate,
Figure 7: Merging environments as needed.
it returns the maximal environment satisfying the system or proves that an inconsistency exists among some constraints.

We have implemented local propagation for the CONSUL language and experimental results have been obtained. Table 1 summarizes some of the results from the experiments. The table lists programs monitored, the sizes of inputs for which data were gathered, the total number of primitive constraints proved or disproved during execution and execution time. The programs shown include a skeletal concurrent database system (Database), two versions of an assignment from an introductory computer science course (Rationals 1 and 2; the original course assignment used Pascal, not CONSUL), a lexical analyzer ( Lexer), and several toy numeric programs (Absolute Value and Vector Sum). These programs range in size from a few lines (Absolute Value) to four or five pages (Database). The experiments suggest that despite its simplicity, local propagation is an adequate base for building real-world constraint languages. This conclusion is admittedly based on a small sample of programs. However, the programs are reasonably varied in style and application, and the results are consistent across them.

In this paper, we have provided a formal basis for local propagation. Based on this work, we believe local propagation a good primary satisfaction method for solving systems of constraints. While more monitoring of constraint programs executed by local propagation is needed, we believe that such monitoring will confirm the relevance of the theoretical results to practical constraint languages.

<table>
<thead>
<tr>
<th>Program</th>
<th>Input Size</th>
<th>Constraints Proved</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute Value</td>
<td>1</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>Absolute Value</td>
<td>5</td>
<td>53</td>
<td>17</td>
</tr>
<tr>
<td>Absolute Value</td>
<td>10</td>
<td>103</td>
<td>27</td>
</tr>
<tr>
<td>Absolute Value</td>
<td>20</td>
<td>203</td>
<td>56</td>
</tr>
<tr>
<td>Vector Sum</td>
<td>1</td>
<td>24</td>
<td>11</td>
</tr>
<tr>
<td>Vector Sum</td>
<td>5</td>
<td>80</td>
<td>23</td>
</tr>
<tr>
<td>Vector Sum</td>
<td>10</td>
<td>150</td>
<td>41</td>
</tr>
<tr>
<td>Vector Sum</td>
<td>20</td>
<td>290</td>
<td>74</td>
</tr>
<tr>
<td>Database</td>
<td>6</td>
<td>1003</td>
<td>174</td>
</tr>
<tr>
<td>Database</td>
<td>16</td>
<td>5171</td>
<td>1046</td>
</tr>
<tr>
<td>Rationals 1</td>
<td>1</td>
<td>487</td>
<td>45</td>
</tr>
<tr>
<td>Rationals 2</td>
<td>1</td>
<td>79</td>
<td>16</td>
</tr>
<tr>
<td>Lexer</td>
<td>2</td>
<td>294</td>
<td>45</td>
</tr>
</tbody>
</table>

Table 1: Statistics on the Number of Constraints Solved by Local Propagation
Acknowledgments

Drafts of this work were usefully critiqued by the students in a seminar on declarative programming languages at the University of Rochester. Many thanks are due to Jerry Feldman and Cesar Quiroz, who have both taken an interest in CONSUL since its beginnings and have made helpful suggestions along the way.
References


